Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Spring & Summer 2024
Lecture 10
2024-4-29

Outline: Intractability

- Computability and computation complexity
- P, NP, and Co-NP
- NP-completeness
- Take-home messages

Outline: Intractability

- Computability and computation complexity
- P, NP, and Co-NP
- NP-completeness
- Take-home messages

Computability

- What kind of problems can be solved by a computer?
- Or more fundamentally:
  - What is computation?
  - Is any math function computable?
  - If a function is computable, how fast?

David Hilbert

Wilhelm Ackermann

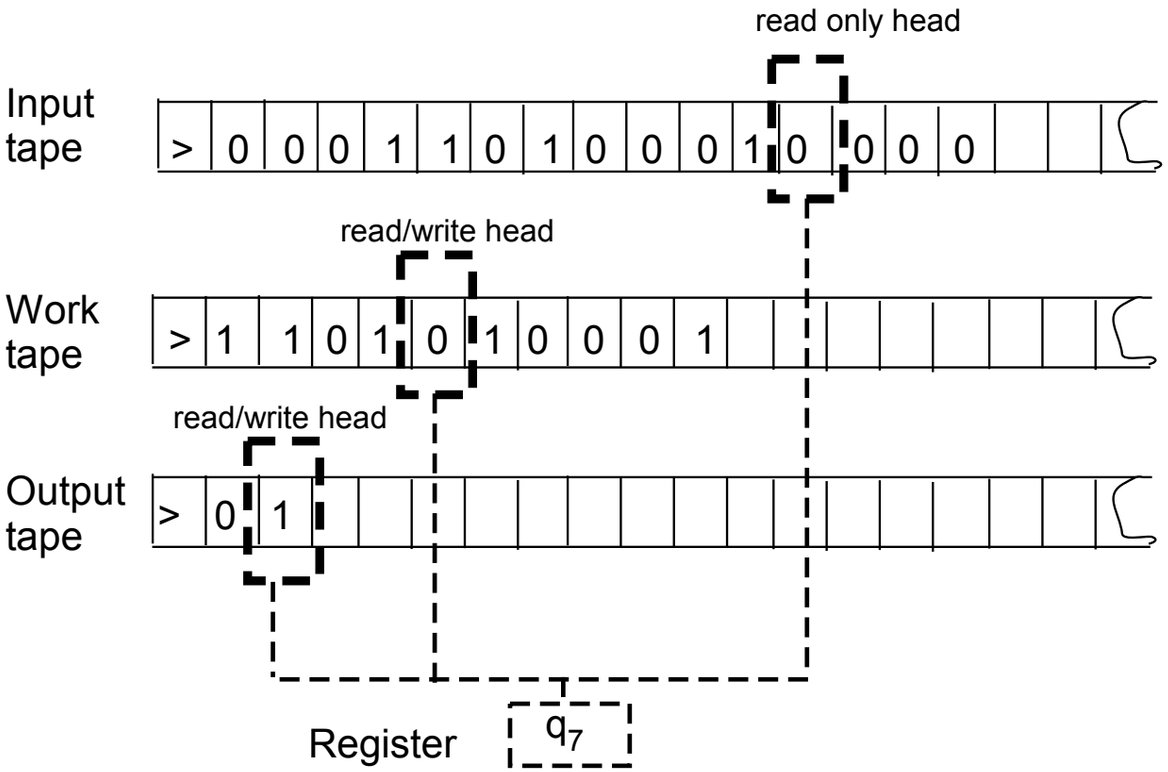
The Entscheidungsproblem (1928):

Find an algorithm to decide whether a given sentence of first-order logic is *true or false*.

Turing Machine (1936)

- Constructed as a tool for showing that the Entscheidungsproblem is not solvable.

- A Turing machine is a function mapping from binary strings $\alpha \in \{0,1\}^*$ to $\{0,1\}$.

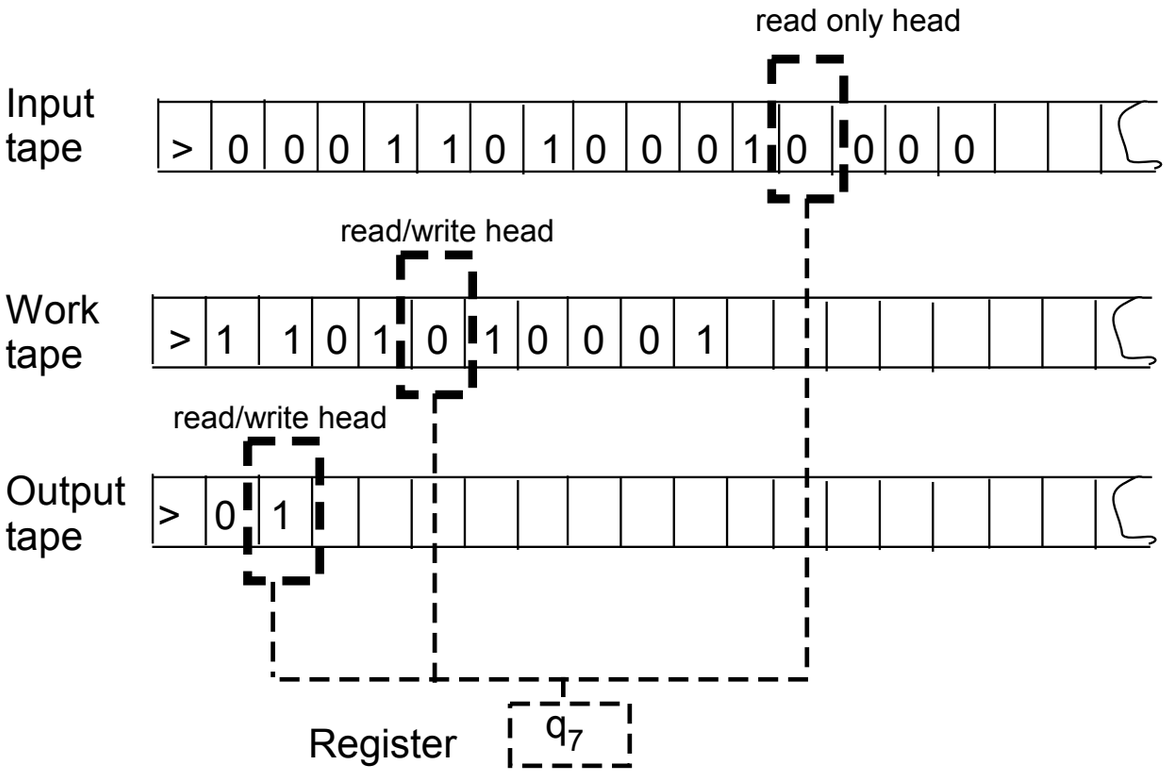


infinite-length tapes (memories)

Turing Machine (1936)

- Constructed as a tool for showing that the Entscheidungsproblem is not solvable.

- A Turing machine is a function mapping from binary strings $\alpha \in \{0,1\}^*$ to $\{0,1\}$.



infinite-length tapes (memories)

Church-Turing Thesis:
Any computable functions can be computed by TMs
(so can Lambda-calculus).

Uncomputable Problems Exist

Theorem:
There exists a function UC: $\{0,1\}^* \to \{0,1\}$ that is not computable by any TM.
Corollary:
The halting problem is not computable by any TM.

Uncomputable Problems Exist

Theorem:
There exists a function UC: $\{0,1\}^* \rightarrow \{0,1\}$ that is not computable by any TM.
Corollary:
The halting problem is not computable by any TM.

- Proof:
  - Any TM can be represented as a binary string.
  - Define UC: for any string α, if $TM_\alpha(\alpha) = 1$, $UC = 0$
    Otherwise $UC = 1$. The Turing machine defined by α
  - Halting is uncomputable: showing that computing UC function can be reduced to solving the halting problem.

Uncomputable Problems Exist

Theorem:
There exists a function UC: $\{0,1\}^* \rightarrow \{0,1\}$ that is not computable by any TM.
Corollary:
The halting problem is not computable by any TM.

- Proof:
  - Any TM can be represented as a binary string.
  - Define UC: for any string α, if $TM_\alpha(\alpha) = 1$, $UC = 0$
    Otherwise $UC = 1$. The Turing machine defined by α
  - Halting is uncomputable: showing that computing UC function can be reduced to solving the halting problem.

Check Arora book pp. 22-23 for details.

Diagonalization

	α_1	α_2	α_3	...
TM_{α_1}	$1 \to 0$ $0 \to 1$			
TM_{α_2}		$1 \to 0$ $0 \to 1$		
TM_{α_3}			$1 \to 0$ $0 \to 1$	
...				

Diagonalization

	α_1	α_2	α_3	...
TM_{α_1}	$1 \to 0$ $0 \to 1$			
TM_{α_2}		$1 \to 0$ $0 \to 1$		
TM_{α_3}			$1 \to 0$ $0 \to 1$	
...				

The UC function tampers all elements in the diagonal.

Diagonalization

	α_1	α_2	α_3	...
TM_{α_1}	$1 \to 0$ $0 \to 1$			
TM_{α_2}		$1 \to 0$ $0 \to 1$		
TM_{α_3}			$1 \to 0$ $0 \to 1$	
...				

The UC function tampers all elements in the diagonal.

The proof technique introduced by Kurt Gödel for proving the incompleteness theorem.

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer:

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

```
Loop( P )
{
/* 1 */ if ( F(P) ) print (YES);
/* 2 */ else infinite_loop();
}
```

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

```
Loop( P )
{
/* 1 */ if ( F(P) ) print (YES);
/* 2 */ else infinite_loop();
}
```

What will happen to Loop(Loop) ?

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

```
Loop( P )
{
/* 1 */ if ( F(P) ) print (YES);
/* 2 */ else infinite_loop();
}
```

What will happen to Loop(Loop) ?

➢ Terminates

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

```
Loop( P )
{
/* 1 */ if ( F(P) ) print (YES);
/* 2 */ else infinite_loop();
}
```

What will happen to Loop(Loop) ?

➢ Terminates ➡ /* 2 */ is true

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

```
Loop( P )
{
/* 1 */ if ( F(P) ) print (YES);
/* 2 */ else infinite_loop();
}
```

What will happen to Loop(Loop) ?

➢ Terminates ➡ /* 2 */ is true ➡ Loops

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

```
Loop( P )
{
/* 1 */ if ( F(P) ) print (YES);
/* 2 */ else infinite_loop();
}
```

What will happen to Loop(Loop) ?

- Terminates → /* 2 */ is true → Loops
- Loops

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

```
Loop( P )
{
/* 1 */ if ( F(P) ) print (YES);
/* 2 */ else infinite_loop();
}
```

What will happen to Loop(Loop) ?

- Terminates → /* 2 */ is true → Loops
- Loops → /* 1 */ is true

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

```
Loop( P )
{
/* 1 */ if ( F(P) ) print (YES);
/* 2 */ else infinite_loop();
}
```

What will happen to Loop(Loop) ?

- Terminates → /* 2 */ is true → Loops
- Loops → /* 1 */ is true → Terminates

【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

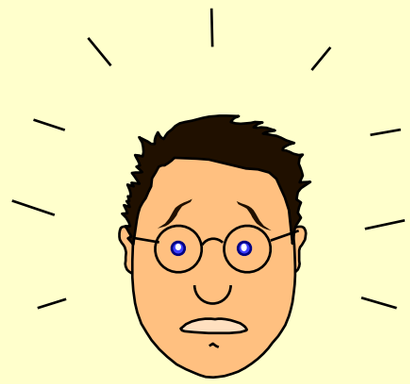
Answer: No.

Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

```
Loop( P )
{
/* 1 */ if ( F(P) ) print (YES);
/* 2 */ else infinite_loop();
}
```

What will happen to Loop(Loop) ?

- Terminates → /* 2 */ is true → Loops
- Loops → /* 1 */ is true → Terminates



【Example】Halting problem: Is it possible to have your C compiler detect all infinite loops?

Answer: No.

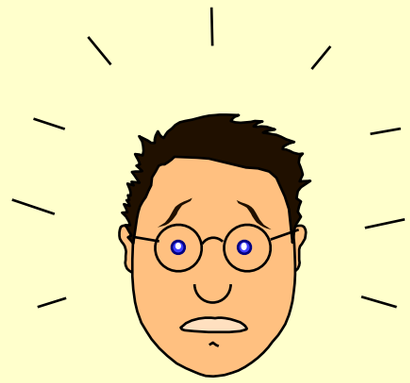
Proof: If there exists an infinite loop-checking program F, we can build the following program and check itself.

Impossible to tell

```
Loop( P )
{
/* 1 */ if ( F(P) ) print (YES);
/* 2 */ else infinite_loop();
}
```

What will happen to Loop(Loop) ?

- Terminates → /* 2 */ is true → Loops
- Loops → /* 1 */ is true → Terminates



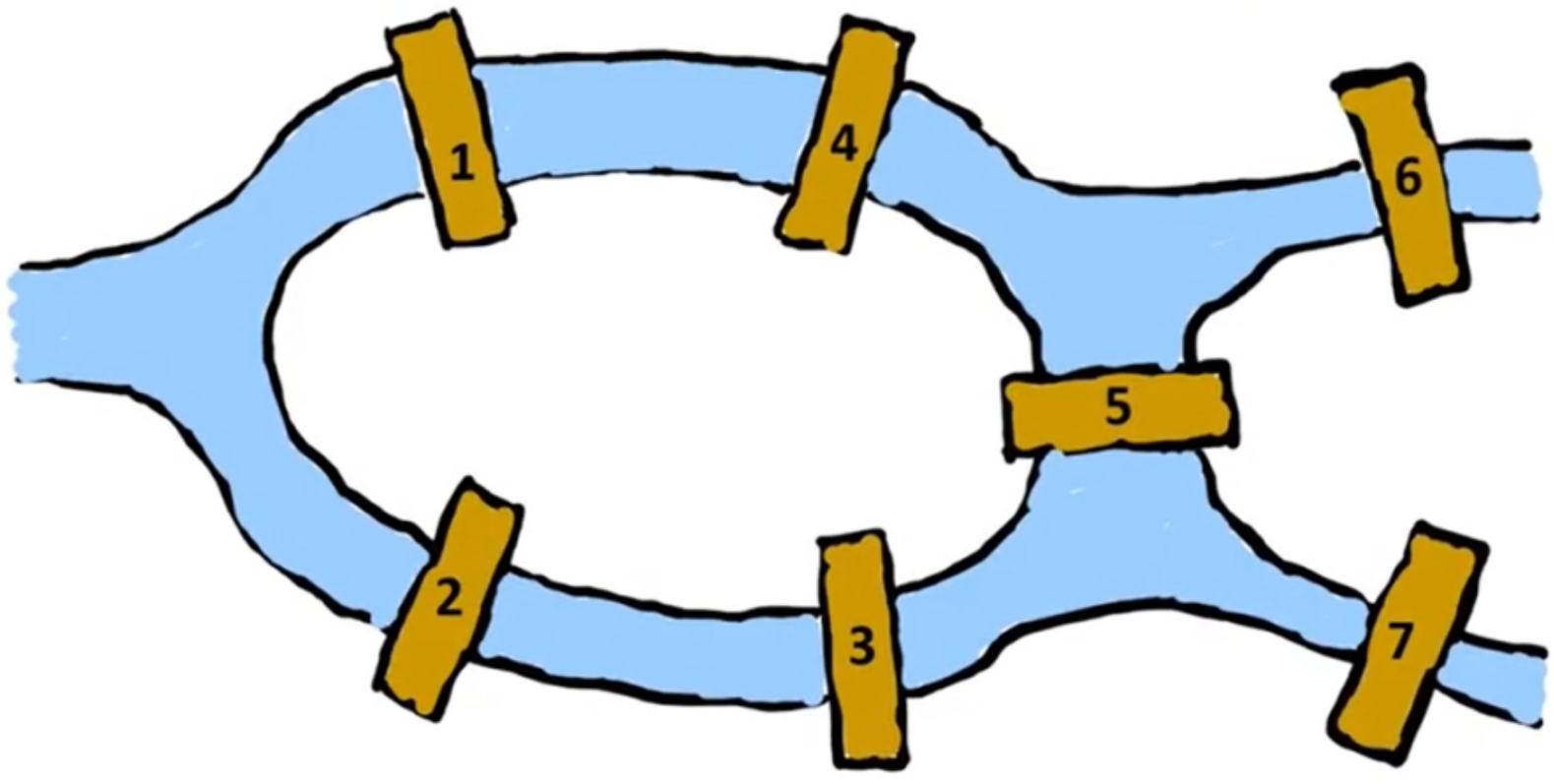
Computational Complexity

- In the previous lectures, not only all the problems are computable, almost all the problems can be solved in polynomial time by TM.

- However, this may not hold for all computable problems:

yes	probably no
shortest path	longest path
min cut	max cut
2-satisfiability	3-satisfiability
planar 4-colorability	planar 3-colorability
bipartite vertex cover	vertex cover
matching	3d-matching
primality testing	factoring
linear programming	integer linear programming

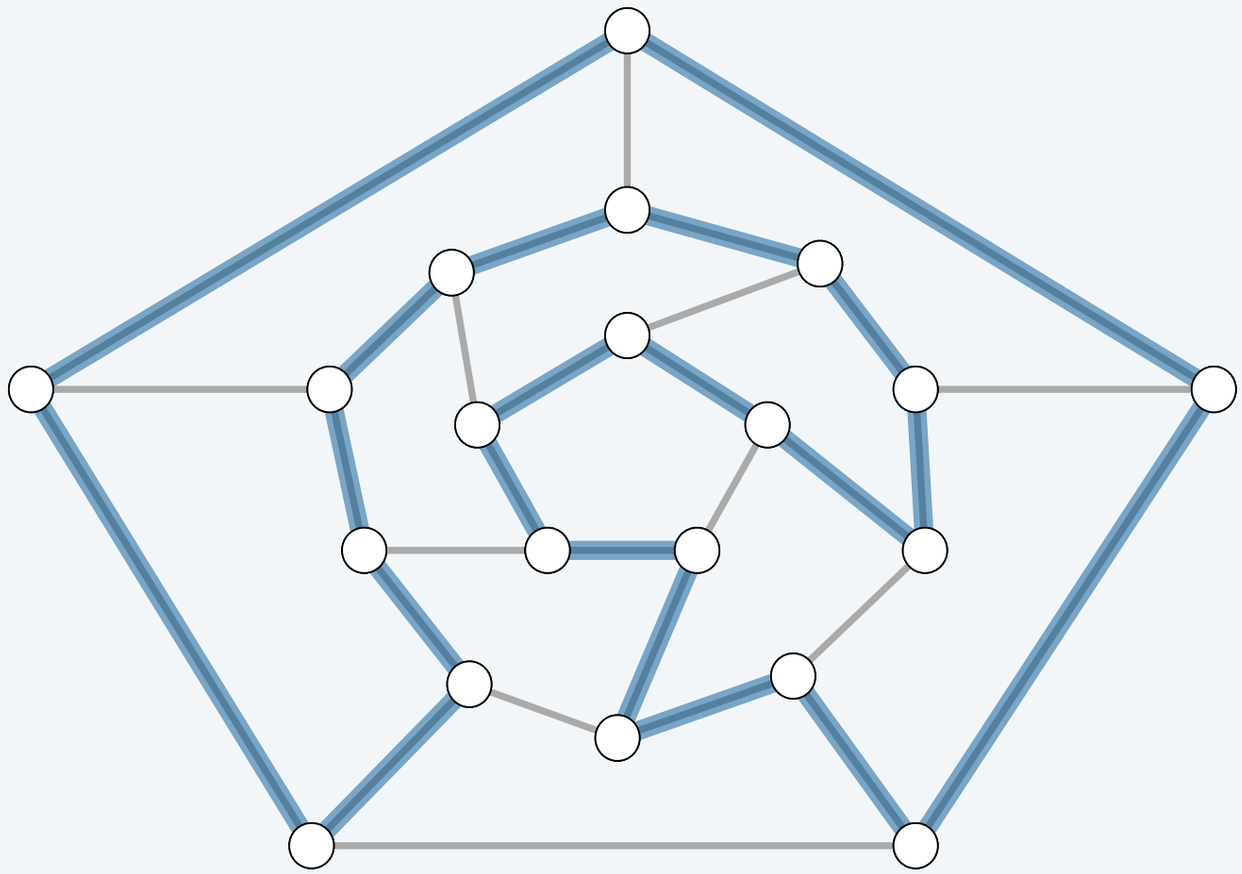
Easy: Finding Euler Cycle



Visiting every edge exactly once.

Hard: Finding Hamiltonian Cycle

HAMILTON-CYCLE. Given an undirected graph $G = (V, E)$, does there exist a cycle Γ that visits every node exactly once?



yes

Visiting every node exactly once.

Satisfiability (SAT) Problem

Literal. A Boolean variable or its negation. x_i or $\overline{x_i}$

Clause. A disjunction of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$

Conjunctive normal form (CNF). A propositional formula Φ that is a conjunction of clauses. $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

Satisfiability (SAT) Problem

Literal. A Boolean variable or its negation. x_i or $\overline{x_i}$

Clause. A disjunction of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$

Conjunctive normal form (CNF). A propositional formula Φ that is a conjunction of clauses. $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

SAT. Given a CNF formula Φ, does it have a satisfying truth assignment?

3-SAT. SAT where each clause contains exactly 3 literals (and each literal corresponds to a different variable).

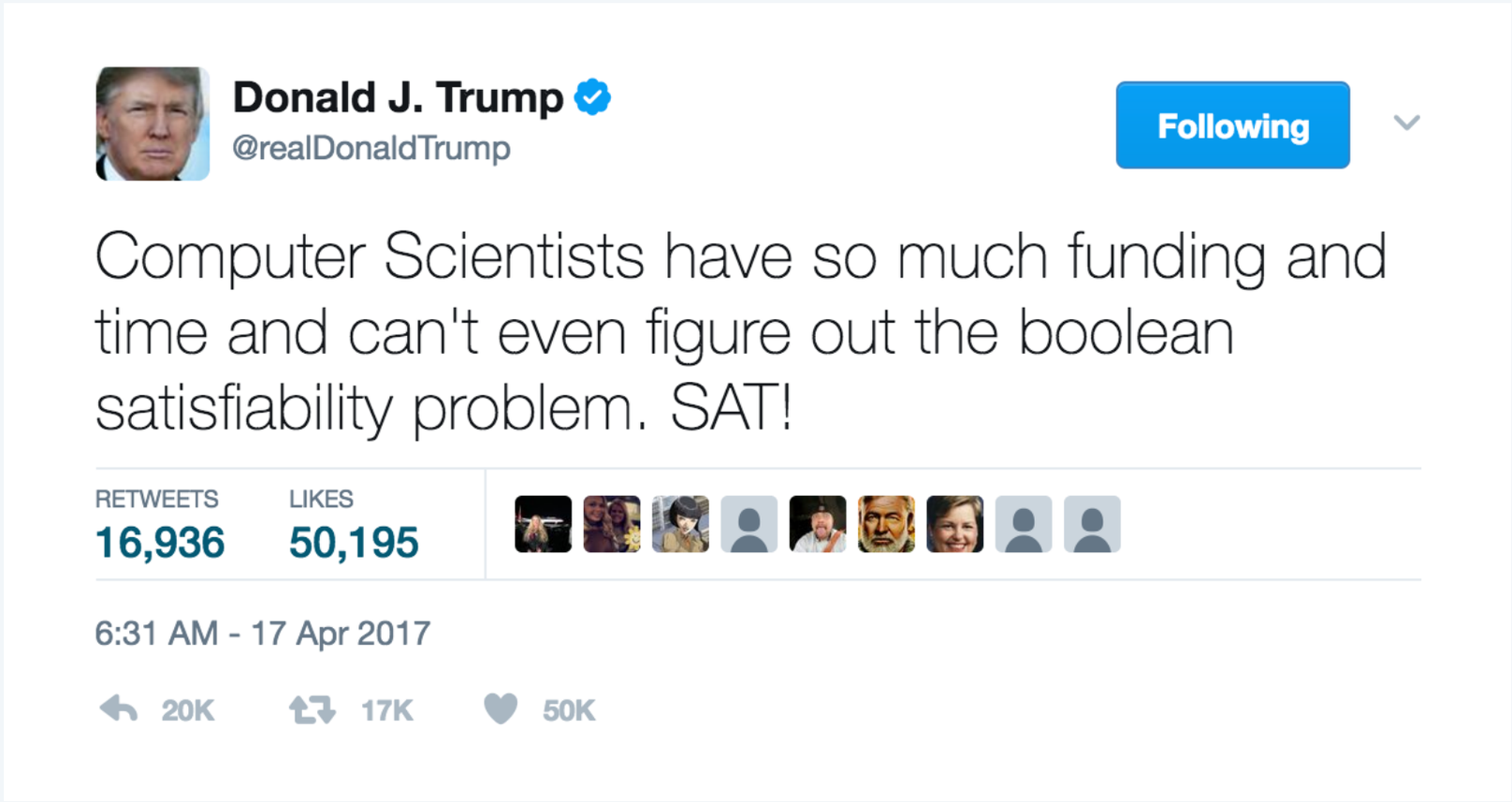
$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3 \right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3 \right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4 \right)$$

yes instance: x_1 = true, x_2 = true, x_3 = false, x_4 = false

Satisfiability is hard

Scientific hypothesis. There does not exist a poly-time algorithm for 3-SAT.

P vs. NP. This hypothesis is equivalent to **P** ≠ **NP** conjecture.

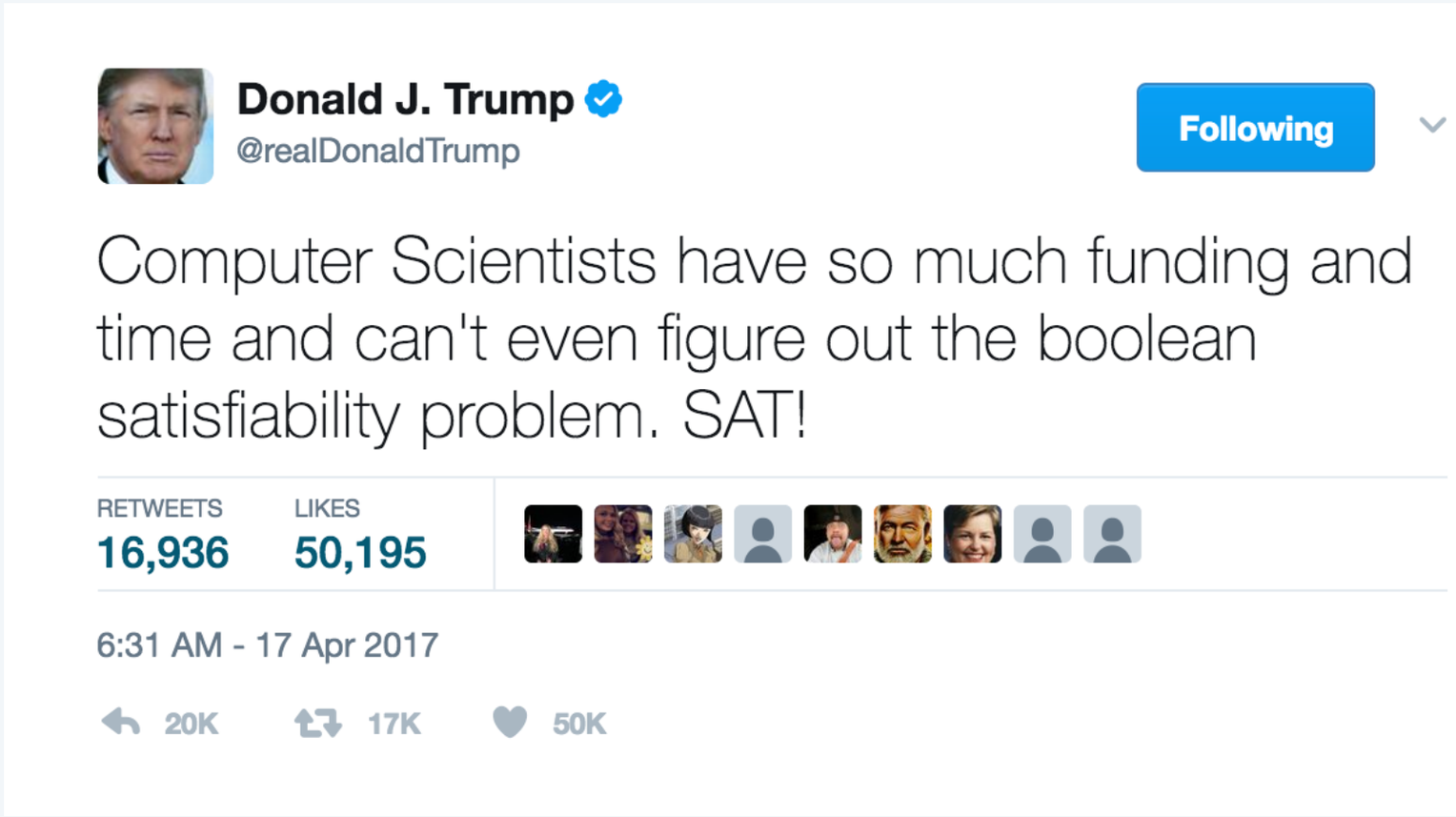


https://www.facebook.com/pg/npcompleteteens

Satisfiability is hard

Scientific hypothesis. There does not exist a poly-time algorithm for 3-SAT.

P vs. NP. This hypothesis is equivalent to **P** ≠ **NP** conjecture.



https://www.facebook.com/pg/npcompleteteens

Note: This is fake news. Just for joke.

Outline: Intractability

- Computability and computation complexity
- P, NP, and Co-NP
- NP-completeness
- Take-home messages

Decision vs. Search Problem

Def. A problem X is called a decision problem if the output is either 0 or 1 (yes/no).

- When we define the P and NP, we only consider decision problems.

Decision vs. Search Problem

Def. A problem X is called a decision problem if the output is either 0 or 1 (yes/no).

- When we define the P and NP, we only consider decision problems.

Fact For each optimization problem X, there is a decision version X' of the problem. If we have a polynomial time algorithm for the decision version X', we can solve the original problem X in polynomial time.

How? Just do binary search over all possible solutions.
More details on DPV Sec. 8.1.

A Formal-language Framework

☞ Abstract Problem

an *abstract problem Q* is a binary relation on a set *I* of problem *instances* and a set *S* of problem *solutions*.

〖Example〗 For SHORTEST-PATH problem

I = { <G, u, v>: G=(V, E) is an undirected graph; u, v ∈ V };

S = { <u, w_1, w_2, …, w_k, v>: <u, w_1>, …, <w_k, v> ∈ E }.

For every $i \in I$, SHORTEST-PATH(i) = $s \in S$.

For decision problem PATH:

I = { <G, u, v, k>: G=(V, E) is an undirected graph; u, v ∈ V;
k ≥ 0 is an integer };

S = { 0, 1 }.

For every $i \in I$, PATH(i) = 1 or 0.

☞ Encodings

Map *I* into a binary string { 0, 1 }* → *Q* is a *concrete problem*.

☞ Formal-language Theory — *for decision problem*

- An *alphabet* Σ is a finite set of symbols $\{0, 1\}$
- A *language* L over Σ is any set of strings made up of symbols from Σ $L = \{ x \in \Sigma^* : Q(x) = 1 \}$
- Denote *empty string* by ε
- Denote *empty language* by $\emptyset$
- Language of all strings over Σ is denoted by Σ^*
- The *complement* of L is denoted by $\Sigma^* - L$
- The *concatenation* of two languages L_1 and L_2 is the language

  $L = \{ x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2 \}$.
- The *closure* or *Kleene star* of a language L is the language

  $L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots$,

  where L^k is the language obtained by concatenating L to itself k times

- **Algorithm A *accepts* a string $x \in \{0, 1\}^*$ if $A(x) = 1$**
- **Algorithm A *rejects* a string x if $A(x) = 0$**
- **A language L is *decided* by an algorithm A if every binary string *in L* is *accepted* by A and every binary string *not in L* is *rejected* by A**
- **To *accept* a language, an algorithm need only worry about strings in L, but to *decide* a language, it must correctly accept or reject every string in $\{0, 1\}^*$**

P = { $L \subseteq \{0, 1\}^*$: there exists an algorithm A that decides L in polynomial time }

- A *verification algorithm* is a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a *certificate*.
- A two-argument algorithm A *verifies* an input string x if there exists a certificate y such that $A(x, y) = 1$.
- The *language* verified by a verification algorithm A is $L = \{ x \in \{0, 1\}^* : \text{there exists } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1 \}$.

〖Example〗 For SAT

$x = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4)$

Certificate: $y = \{ x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1 \}$

A language L belongs to NP iff there exist a two-input polynomial-time algorithm A and a constant c such that $L = \{ x \in \{0, 1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1 \}$. We say that algorithm A *verifies* language L in *polynomial time*.

Encoding of Decision Problems

string from a language

The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 1111011111000111111000011000001 1100001101111111111000001

Encoding of Decision Problems

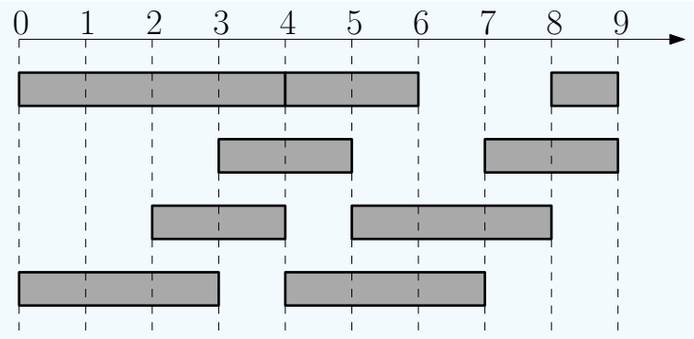
string from a language

The input of a problem will be encoded as a binary string.

Example: Sorting problem

- Input: (3, 6, 100, 9, 60)
- Binary: (11, 110, 1100100, 1001, 111100)
- String: 1111011111000111111000011000001 1100001101111111111000001

Example: Interval Scheduling Problem



- $(0, 3, 0, 4, 2, 4, 3, 5, 4, 6, 4, 7, 5, 8, 7, 9, 8, 9)$
- Encode the sequence into a binary string as before

The Class P

Def. A decision problem X is a function mapping $\{0,1\}^*$ to $\{0,1\}$ such that for any $s \in \{0,1\}^*$, $X(s)$ is the correct output for input s.

- $\{0,1\}^*$: the set of all binary strings of any length.

The Class P

Def. A decision problem X is a function mapping $\{0,1\}^*$ to $\{0,1\}$ such that for any $s \in \{0,1\}^*$, $X(s)$ is the correct output for input s.

- $\{0,1\}^*$: the set of all binary strings of any length.

Exactly what a Turing machine does.

The Class P

Def. A decision problem X is a function mapping $\{0,1\}^*$ to $\{0,1\}$ such that for any $s \in \{0,1\}^*$, $X(s)$ is the correct output for input s.

- $\{0,1\}^*$: the set of all binary strings of any length.

Exactly what a Turing machine does.

Def. An algorithm A solves a problem X if, $A(s) = X(s)$ for any binary string s

Def. A has a polynomial running time if there is a polynomial function $p(\cdot)$ so that for every string s, the algorithm A terminates on s in at most $p(|s|)$ steps.

The Class P

Def. The complexity class P is the set of decision problems X that can be solved in polynomial time.

- The decision versions of interval scheduling, shortest path and minimum spanning tree all in P.

The Class NP

- Consider Hamiltonian cycle problem (HC).
  - Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
  - Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that G contains a Hamiltonian cycle?

The Class NP

- Consider Hamiltonian cycle problem (HC).
  - Alice has a supercomputer, fast enough to run the $2^{O(n)}$ time algorithm for HC
  - Bob has a slow computer, which can only run an $O(n^3)$-time algorithm

Q: Given a graph $G = (V, E)$ with a HC, how can Alice convince Bob that G contains a Hamiltonian cycle?

A: Alice gives a Hamiltonian cycle to Bob, and Bob checks if it is really a Hamiltonian cycle of G

Def. The message Alice sends to Bob is called a certificate, and the algorithm Bob runs is called a certifier.

The Class NP

Def. B is an efficient certifier for a problem X if

- B is a polynomial-time algorithm that takes two input strings s and t, and outputs 0 or 1.
- there is a polynomial function p such that, $X(s) = 1$ if and only if there is string t such that $|t| \leq p(|s|)$ and $B(s, t) = 1$.

The string t such that $B(s, t) = 1$ is called a certificate.

The Class NP

Def. B is an efficient certifier for a problem X if

- B is a polynomial-time algorithm that takes two input strings s and t, and outputs 0 or 1.
- there is a polynomial function p such that, $X(s) = 1$ if and only if there is string t such that $|t| \leq p(|s|)$ and $B(s, t) = 1$.

The string t such that $B(s, t) = 1$ is called a certificate.

Def. The complexity class NP is the set of all problems for which there exists an efficient certifier.

The Class NP

Def. B is an efficient certifier for a problem X if

- B is a polynomial-time algorithm that takes two input strings s and t, and outputs 0 or 1.
- there is a polynomial function p such that, $X(s) = 1$ if and only if there is string t such that $|t| \leq p(|s|)$ and $B(s, t) = 1$.

The string t such that $B(s, t) = 1$ is called a certificate.

Def. The complexity class NP is the set of all problems for which there exists an efficient certifier.

Given solution t for problem X with input s,
The algorithm B can check if the answer t is correct in poly-time.

The Class NP

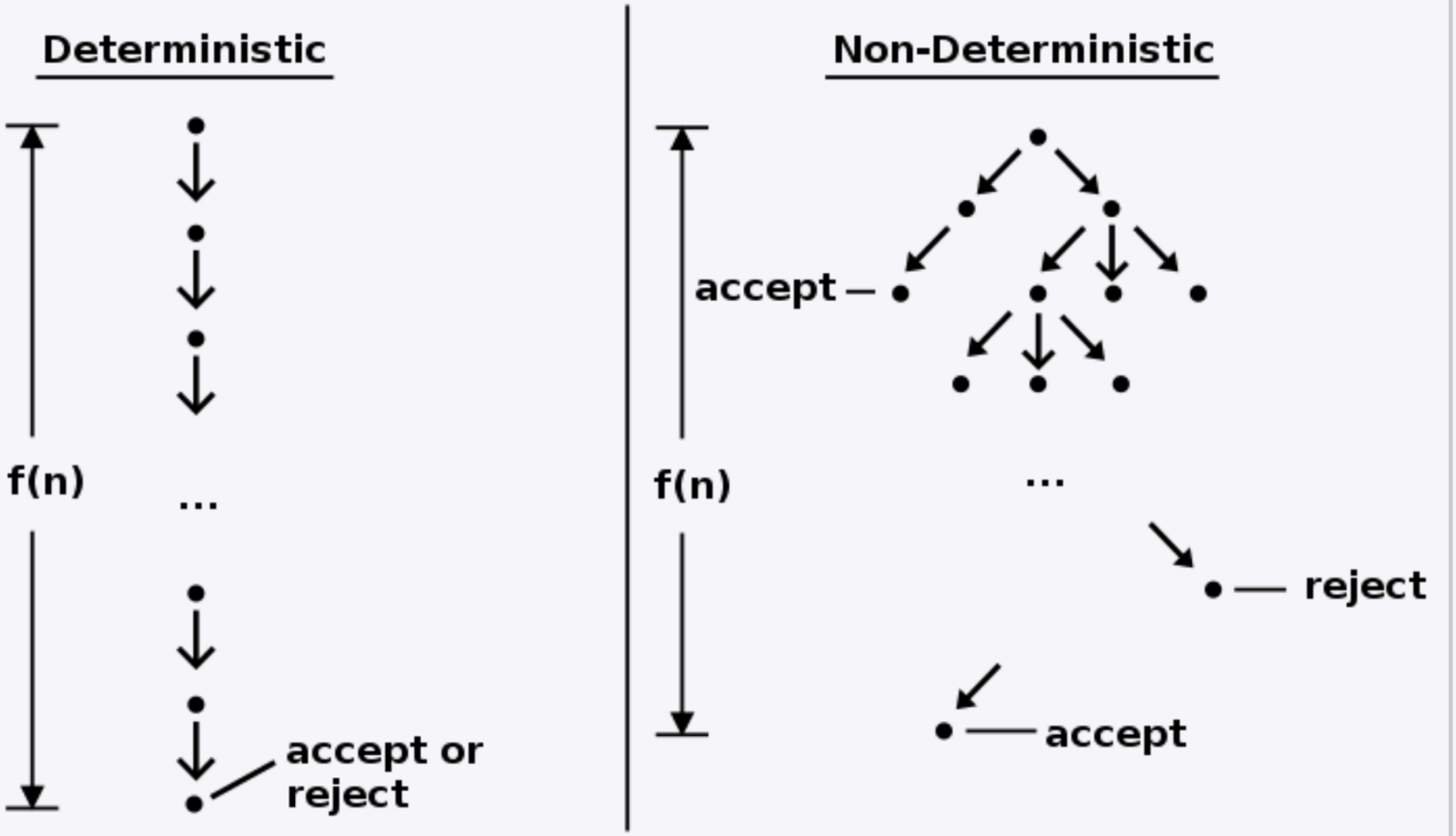
- Input: Graph G
- Certificate: a permutation S of V that forms a Hamiltonian Cycle
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function p
- Certifier B: $B(G, S) = 1$ if and only if S gives an HC in G
- Clearly, B runs in polynomial time
- $\text{HC}(G) = 1 \iff \exists S,\ B(G, S) = 1$

The Class NP

- Input: Graph G
- Certificate: a permutation S of V that forms a Hamiltonian Cycle
- $|\text{encoding}(S)| \leq p(|\text{encoding}(G)|)$ for some polynomial function p
- Certifier B: $B(G, S) = 1$ if and only if S gives an HC in G
- Clearly, B runs in polynomial time
- $\text{HC}(G) = 1 \iff \exists S,\ B(G, S) = 1$

Hamiltonian cycle problem (HC) is in NP.

Nondeterministic Turing Machine



Comparison of deterministic and nondeterministic computation

Can have different actions for each state, leading to different final results.
Having the power of verifying a certificate.
NP: Nondeterministic Poly-time

The Class Co-NP

$\overline{\text{HC}}$

Input: graph $G = (V, E)$

Output: whether G does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in$ NP?
- Can Alice convince Bob that G is a yes-instance (i.e, G does not contain a HC), if this is true.
- Unlikely

The Class Co-NP

$\overline{\text{HC}}$

Input: graph $G = (V, E)$

Output: whether G does not contain a Hamiltonian cycle

- Is $\overline{\text{HC}} \in$ NP?
- Can Alice convince Bob that G is a yes-instance (i.e, G does not contain a HC), if this is true.
- Unlikely

- Alice can only convince Bob that G is a no-instance
- $\overline{\text{HC}} \in$ Co-NP

The Class Co-NP

Def. For a problem X, the problem $\overline{X}$ is the problem such that $\overline{X}(s) = 1$ if and only if $X(s) = 0$.

Def. Co-NP is the set of decision problems X such that $\overline{X} \in$ NP.

The Class Co-NP

Def. A tautology is a boolean formula that always evaluates to 1.

Tautology Problem

Input: a boolean formula

Output: whether the formula is a tautology

- e.g. $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3)$ is a tautology
- Bob can certify that a formula is not a tautology
- Thus Tautology $\in$ Co-NP

The Class Co-NP

Def. A tautology is a boolean formula that always evaluates to 1.

Tautology Problem

Input: a boolean formula

Output: whether the formula is a tautology

- e.g. $(\neg x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_3) \vee x_1 \vee (\neg x_2 \wedge x_3)$ is a tautology
- Bob can certify that a formula is not a tautology
- Thus Tautology $\in$ Co-NP

What's the relationship among P, NP, and Co-NP?

Letter from Gödel to von Neumann (1956)

Since you now, as I hear, are feeling stronger, I would like to allow myself to write you about a mathematical problem, of which your opinion would very much interest me: One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F,n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max_F \psi(F,n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi(n) \geq k \cdot n$. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\varphi(n)$ grows that slowly. Since

1. it seems that $\varphi(n) \geq k \cdot n$ is the only estimation which one can obtain by a generalization of the proof of the undecidability of the Entscheidungsproblem and
2. after all $\varphi(n) \sim k \cdot n$ (or $\sim k \cdot n^2$) only means that the number of steps as opposed to trial and error can be reduced from N to log N (or $(\log N)^2$).

However, such strong reductions appear in other finite problems, for example in the computation of the quadratic residue symbol using repeated application of the law of reciprocity. It would be interesting to know, for instance, the situation concerning the determination of primality of a number and how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search.

The P vs NP Problem

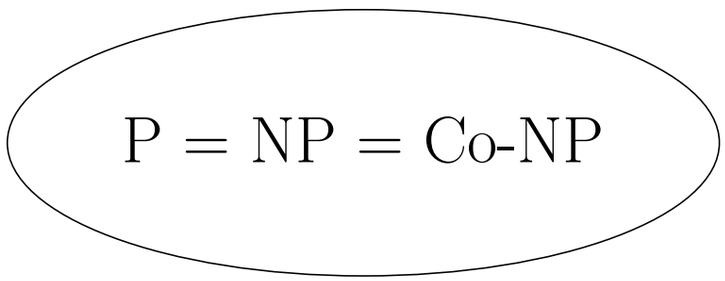
- P = NP?
  - A famous, big, and fundamental open problem in computer science
  - Little progress has been made
  - Most researchers believe P $\neq$ NP
  - It would be too amazing if P = NP: if one can check a solution efficiently, then one can find a solution efficiently
  - We assume P $\neq$ NP and prove that problems do not have polynomial time algorithms.
  - We said it is unlikely that Hamiltonian Cycle can be solved in polynomial time:
    - if P $\neq$ NP, then HC $\notin$ P
    - HC $\notin$ P, unless P = NP

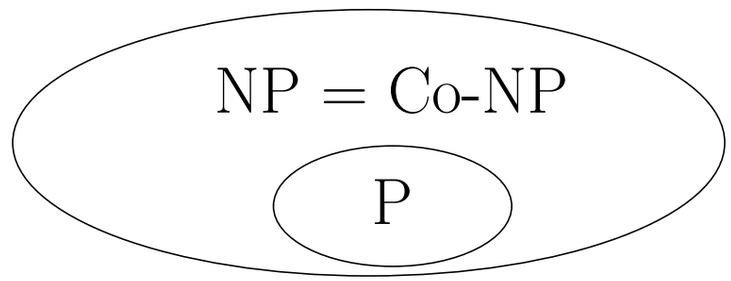
The P vs NP Problem

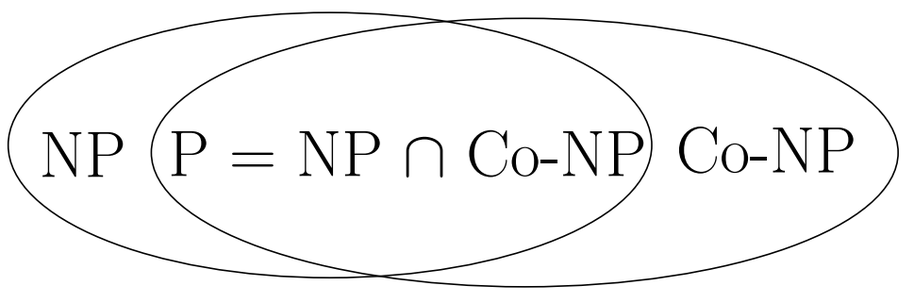
- NP = Co-NP?
  - Again, a big open problem
  - Most researchers believe NP $\neq$ Co-NP.

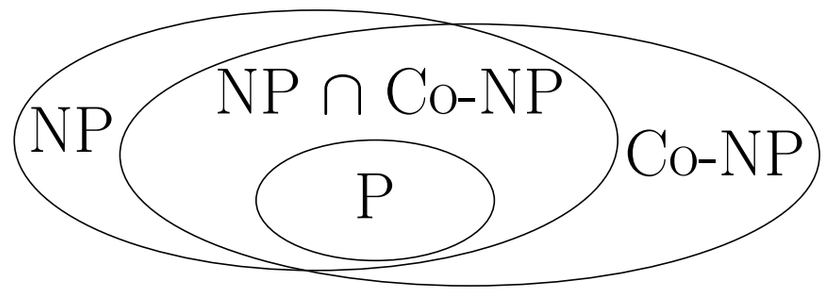
4 Possibilities of Relationships

Notice that $X \in$ NP $\iff \overline{X} \in$ Co-NP and P $\subseteq$ NP $\cap$ Co-NP









- People commonly believe we are in the 4th scenario

Possible outcomes

P ≠ NP

“ I conjecture that there is no good algorithm for the traveling salesman problem. My reasons are the same as for any mathematical conjecture: (i) It is a legitimate mathematical possibility and (ii) I do not know.”

— *Jack Edmonds 1966*

“ In my view, there is no way to even make intelligent guesses about the answer to any of these questions. If I had to bet now, I would bet that P is not equal to NP. I estimate the half-life of this problem at 25–50 more years, but I wouldn’t bet on it being solved before 2100. ”

— *Bob Tarjan (2002)*

Possible outcomes

P ≠ NP

> *" We seem to be missing even the most basic understanding of the nature of its difficulty…. All approaches tried so far probably (in some cases, provably) have failed. In this sense P =NP is different from many other major mathematical problems on which a gradual progress was being constantly done (sometimes for centuries) whereupon they yielded, either completely or partially. "*
>
> — *Alexander Razborov (2002)*

Possible outcomes

P = NP

> *“ I think that in this respect I am on the loony fringe of the mathematical community: I think (not too strongly!) that P=NP and this will be proved within twenty years. Some years ago, Charles Read and I worked on it quite bit, and we even had a celebratory dinner in a good restaurant before we found an absolutely fatal mistake. ”*
>
> — *Béla Bollobás (2002)*

> *“ In my opinion this shouldn’t really be a hard problem; it’s just that we came late to this theory, and haven’t yet developed any techniques for proving computations to be hard. Eventually, it will just be a footnote in the books. ”* — *John Conway*

Other possible outcomes

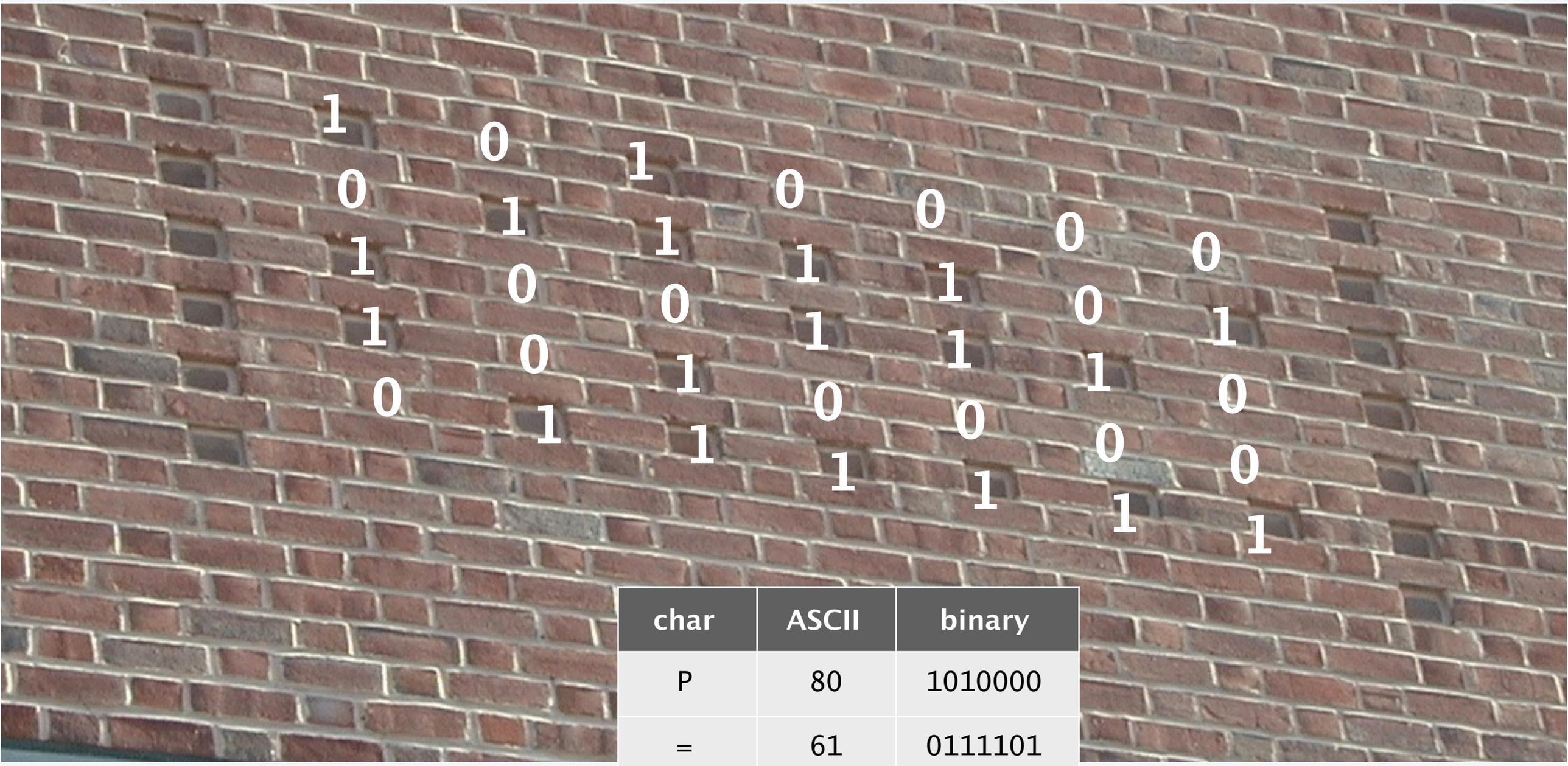
P = **NP**, but only $\Omega(n^{100})$ algorithm for 3-SAT.

P ≠ **NP**, but with $O(n^{\log^* n})$ algorithm for 3-SAT.

P = **NP** is independent (of ZFC axiomatic set theory).

" It will be solved by either 2048 or 4096. I am currently somewhat pessimistic. The outcome will be the truly worst case scenario: namely that someone will prove P = NP because there are only finitely many obstructions to the opposite hypothesis; hence there exists a polynomial time solution to SAT but we will never know its complexity! " — *Donald Knuth*

Princeton CS Building, West Wall, Circa 2001



char	ASCII	binary
P	80	1010000
=	61	0111101
N	78	1001110
P	80	1010000
?	63	0111111

Outline: Intractability

- Computability and computation complexity
- P, NP, and Co-NP
- NP-completeness
- Take-home messages

Proofs of impossibility were effected by the ancients ... [and] in later mathematics, the question as to the impossibility of certain solutions plays a preminent part. ...

In other sciences also one meets old problems which have been settled in a manner most satisfactory and most useful to science by the proof of their impossibility. ... After seeking in vain for the construction of a perpetual motion machine, the relations were investigated which must subsist between the forces of nature if such a machine is to be impossible; and this inverted question led to the discovery of the law of the conservation of energy. ...

It is probably this important fact along with other philosophical reasons that gives rise to conviction ... that every definite mathematical problem must necessary be susceptible of an exact settlement, either in the form of an actual answer to the question asked, or by the proof of the impossibility of its solution and therewith the necessary failure of all attempts. ... This conviction... is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorabimus.

David Hilbert, Lecture delivered before the International Congress of Mathematicians at Paris in 1900

NP-Completeness

- In previous lectures, we focus on design efficient algorithms for solving various of computation problems.

- NP-completeness provides negative results: some problems are not likely to be solved efficiently.

- Why do we study negative results?
  - Understanding the limitation of the ability of computers.
  - Avoiding useless efforts in algorithm design.

Poly-Time Reductions

Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

Poly-Time Reductions

Def. Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

To prove positive results:

Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can be solved in polynomial time.

To prove negative results:

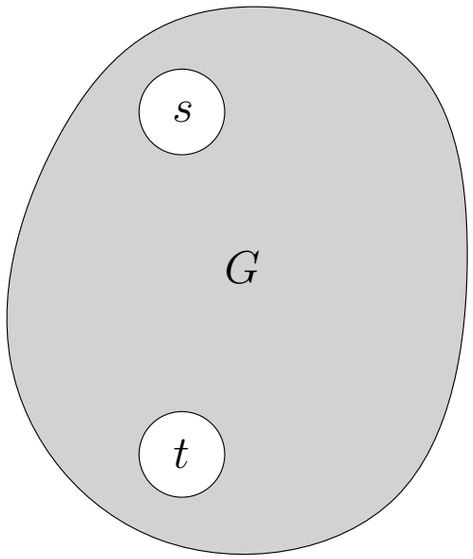
Suppose $Y \leq_P X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.

Reduction from Hamiltonian Path to Cycle

Hamiltonian-Path (HP) problem

Input: $G = (V, E)$ and $s, t \in V$

Output: whether there is a Hamiltonian path from s to t in G

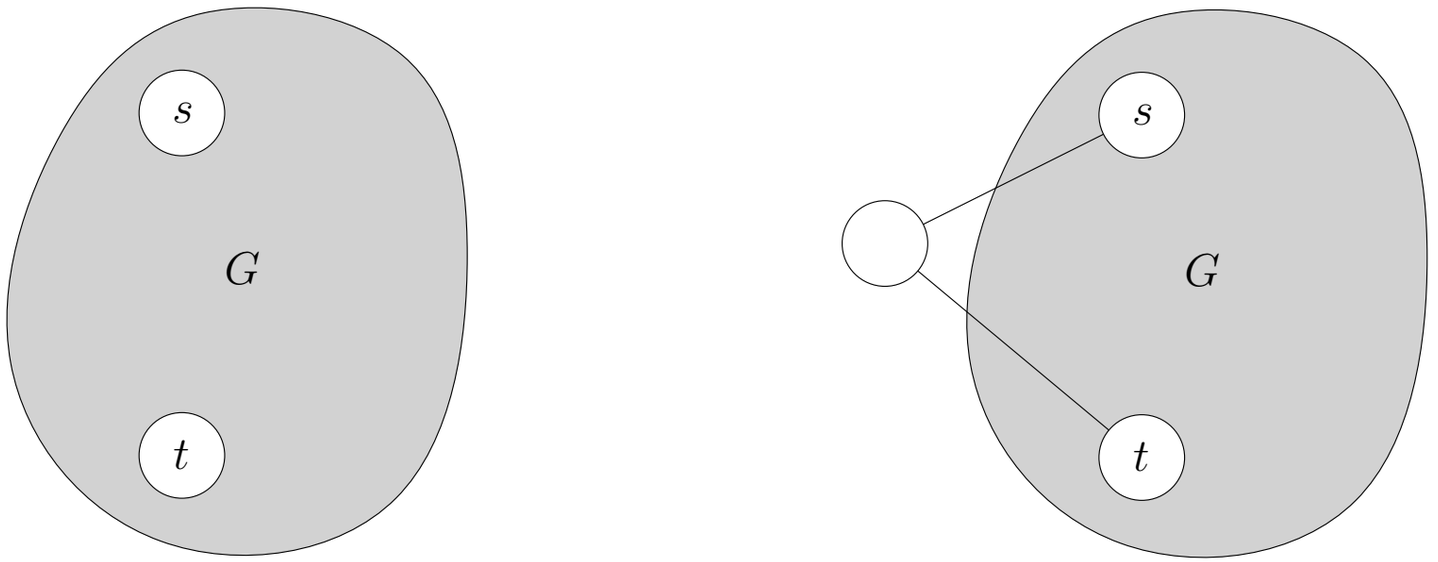


Reduction from Hamiltonian Path to Cycle

Hamiltonian-Path (HP) problem

Input: $G = (V, E)$ and $s, t \in V$

Output: whether there is a Hamiltonian path from s to t in G

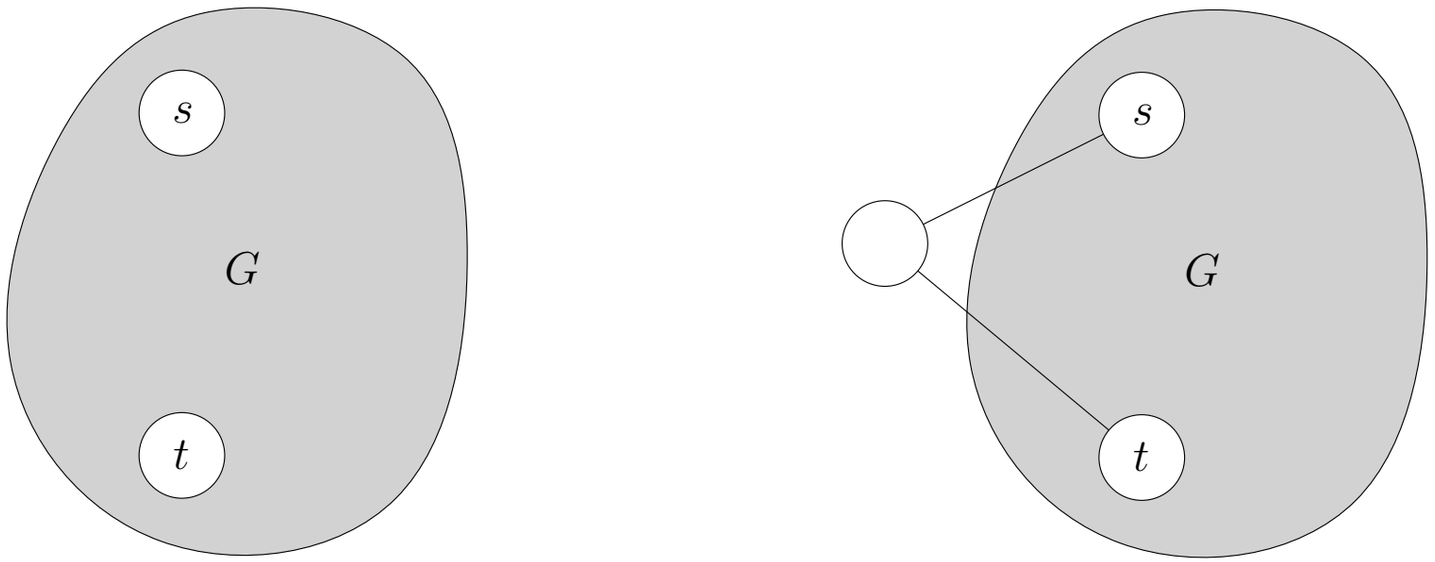


Reduction from Hamiltonian Path to Cycle

Hamiltonian-Path (HP) problem

Input: $G = (V, E)$ and $s, t \in V$

Output: whether there is a Hamiltonian path from s to t in G



G has a HP from s to t if and only if the right graph has a HC.

$$HP \leq_P HC$$

NP-Complete and NP-Hard

Def. A problem X is called NP-complete if

1. $X \in$ NP, and
2. $Y \leq_{\mathsf{P}} X$ for every $Y \in$ NP.

NP-Complete and NP-Hard

Def. A problem X is called NP-complete if

1. $X \in$ NP, and
2. $Y \leq_{\mathsf{P}} X$ for every $Y \in$ NP.

Theorem If X is NP-complete and $X \in$ P, then P $=$ NP.

- NP-complete problems are the hardest problems in NP
- NP-hard problems are at least as hard as NP-complete problems (a NP-hard problem is not required to be in NP)

NP-Complete Problems

Def. A problem X is called NP-complete if

1. $X \in$ NP, and
2. $Y \leq_P X$ for every $Y \in$ NP.

- How can we find a problem $X \in$ NP such that every problem $Y \in$ NP is polynomial time reducible to X? Are we asking for too much?
- No! There is indeed a large family of natural NP-complete problems

The "first" NP-complete problem

Theorem. [Cook 1971, Levin 1973] SAT ∈ **NP**-complete.

The Complexity of Theorem-Proving Procedures

Stephen A. Cook

University of Toronto

Summary

It is shown that any recognition problem solved by a polynomial time-bounded nondeterministic Turing machine can be "reduced" to the problem of determining whether a given propositional formula is a tautology. Here "reduced" means, roughly speaking, that the first problem can be solved deterministically in polynomial time provided an oracle is available for solving the second. From this notion of reducible, polynomial degrees of difficulty are defined, and it is shown that the problem of determining tautologyhood has the same polynomial degree as the problem of determining whether the first of two given graphs is isomorphic to a subgraph of the second. Other examples are discussed. A method of measuring the complexity of proof procedures for the predicate calculus is introduced and discussed.

Throughout this paper, a set of strings means a set of strings on some fixed, large, finite alphabet Σ. This alphabet is large enough to include symbols for all sets described here. All Turing machines are deterministic recognition devices, unless the contrary is explicitly stated.

1. Tautologies and Polynomial Reducibility.

Let us fix a formalism for the propositional calculus in which formulas are written as strings on Σ. Since we will require infinitely many proposition symbols (atoms), each such symbol will consist of a member of Σ followed by a number in binary notation to distinguish that symbol. Thus a formula of length n can only have about n/logn distinct function and predicate symbols. The logical connectives are & (and), ∨ (or), and ¬(not).

The set of tautologies (denoted by {tautologies}) is a certain recursive set of strings on this alphabet, and we are interested in the problem of finding a good lower bound on its possible recognition times. We provide no such lower bound here, but theorem 1 will give evidence that {tautologies} is a difficult set to recognize, since many apparently difficult problems can be reduced to determining tautologyhood. By reduced we mean, roughly speaking, that if tautologyhood could be decided instantly (by an "oracle") then these problems could be decided in polynomial time. In order to make this notion precise, we introduce query machines, which are like Turing machines with oracles in [1].

A query machine is a multitape Turing machine with a distinguished tape called the query tape, and three distinguished states called the query state, yes state, and no state, respectively. If M is a query machine and T is a set of strings, then a T-computation of M is a computation of M in which initially M is in the initial state and has an input string w on its input tape, and each time M assumes the query state there is a string u on the query tape, and the next state M assumes is the yes state if u∈T and the no state if u∉T. We think of an "oracle", which knows T, placing M in the yes state or no state.

Definition

A set S of strings is P-reducible (P for polynomial) to a set T of strings iff there is some query machine M and a polynomial Q(n) such that for each input string w, the T-computation of M with input w halts within Q(|w|) steps (|w| is the length of w), and ends in an accepting state iff w∈S.

It is not hard to see that P-reducibility is a transitive relation. Thus the relation E on

ПРОБЛЕМЫ ПЕРЕДАЧИ ИНФОРМАЦИИ

Том IX *1973* *Вып. 3*

КРАТКИЕ СООБЩЕНИЯ

УДК 519.14

УНИВЕРСАЛЬНЫЕ ЗАДАЧИ ПЕРЕБОРА

Л. А. Левин

В статье рассматривается несколько известных массовых задач «переборного типа» и доказывается, что эти задачи можно решать лишь за такое время, за которое можно решать вообще любые задачи указанного типа.

После уточнения понятия алгоритма была доказана алгоритмическая неразрешимость ряда классических массовых проблем (например, проблем тождества элементов групп, гомеоморфности многообразий, разрешимости диофантовых уравнений и других). Тем самым был снят вопрос о нахождении практического способа их решения. Однако существование алгоритмов для решения других задач не снимает для них аналогичного вопроса из-за фантастически большого объема работы, предписываемого этими алгоритмами. Такова ситуация с так называемыми переборными задачами: минимизации булевых функций, поиска доказательств ограниченной длины, выяснения изоморфности графов и другими. Все эти задачи решаются тривиальными алгоритмами, состоящими в переборе всех возможностей. Однако эти алгоритмы требуют экспоненциального времени работы и у математиков сложилось убеждение, что более простые алгоритмы для них невозможны. Был получен ряд серьезных аргументов в пользу его справедливости (см.[1, 2]), однако доказать это утверждение не удалось никому. (Например, до сих пор не доказано, что для нахождения математических доказательств нужно больше времени, чем для их проверки.)

Однако если предположить, что вообще существует какая-нибудь (хотя бы искусственно построенная) массовая задача переборного типа, неразрешимая простыми (в смысле объема вычислений) алгоритмами, то можно показать, что этим же свойством обладают и многие «классические» переборные задачи (в том числе задача минимизации, задача поиска доказательств и др.). В этом и состоят основные результаты статьи.

Функции $f(n)$ и $g(n)$ будем называть сравнимыми, если при некотором k

$$f(n) \leqslant (g(n)+2)^k \quad \text{и} \quad g(n) \leqslant (f(n)+2)^k.$$

Аналогично будем понимать термин «меньше или сравнимо».

О п р е д е л е н и е. Задачей переборного типа (или просто переборной задачей) будем называть задачу вида «по данному x найти какое-нибудь y длины, сравнимой с длиной x, такое, что выполняется $A(x, y)$», где $A(x, y)$ — какое-нибудь свойство, проверяемое алгоритмом, время работы которого сравнимо с длиной x. (Под алгоритмом здесь можно понимать, например, алгоритмы Колмогорова — Успенского или машины Тьюринга, или нормальные алгоритмы; x, y — двоичные слова). Квазипереборной задачей будем называть задачу выяснения, существует ли такое y.

Мы рассмотрим шесть задач этих типов. Рассматриваемые в них объекты кодируются естественным образом в виде двоичных слов. При этом выбор естественной кодировки не существен, так как все они дают сравнимые длины кодов.

Задача 1. Заданы списком конечное множество и покрытие его 500-элементными подмножествами. Найти подпокрытие заданной мощности (соответственно выяснить существует ли оно).

Задача 2. Таблично задана частичная булева функция. Найти заданного размера дизъюнктивную нормальную форму, реализующую эту функцию в области определения (соответственно выяснить существует ли она).

Задача 3. Выяснить, выводима или опровержима данная формула исчисления высказываний. (Или, что то же самое, равна ли константе данная булева формула.)

Задача 4. Даны два графа. Найти гомоморфизм одного на другой (выяснить его существование).

Задача 5. Даны два графа. Найти изоморфизм одного в другой (на его часть).

Задача 6. Рассматриваются матрицы из целых чисел от 1 до 100 и некоторое условие о том, какие числа в них могут соседствовать по вертикали и какие по горизонтали. Заданы числа на границе и требуется продолжить их на всю матрицу с соблюдением условия.

Karp's 20 poly-time reductions from satisfiability

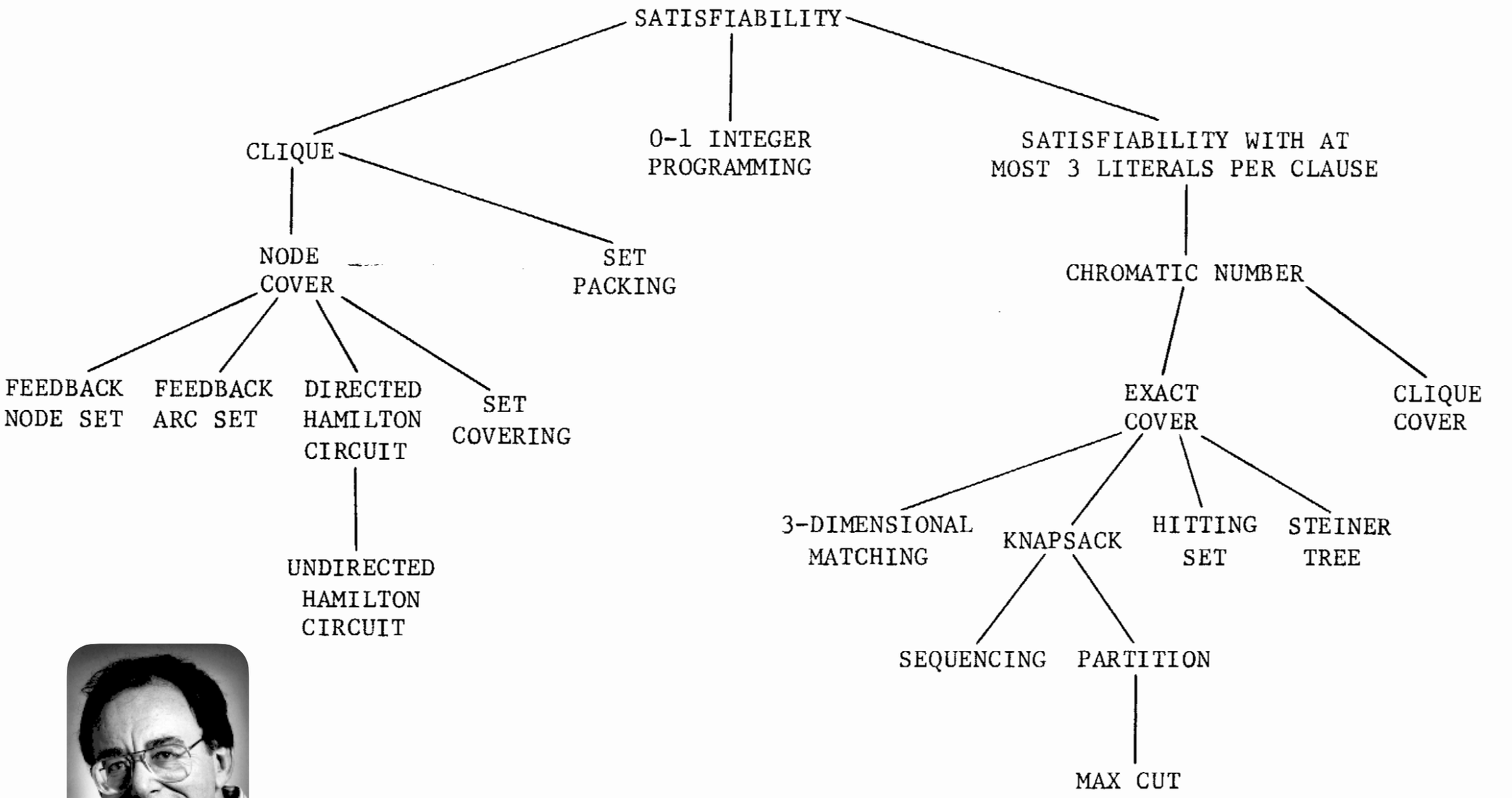


FIGURE 1 - Complete Problems

RICHARD M. KARP

Dick Karp (1972)
1985 Turing Award

Karp's 20 poly-time reductions from satisfiability

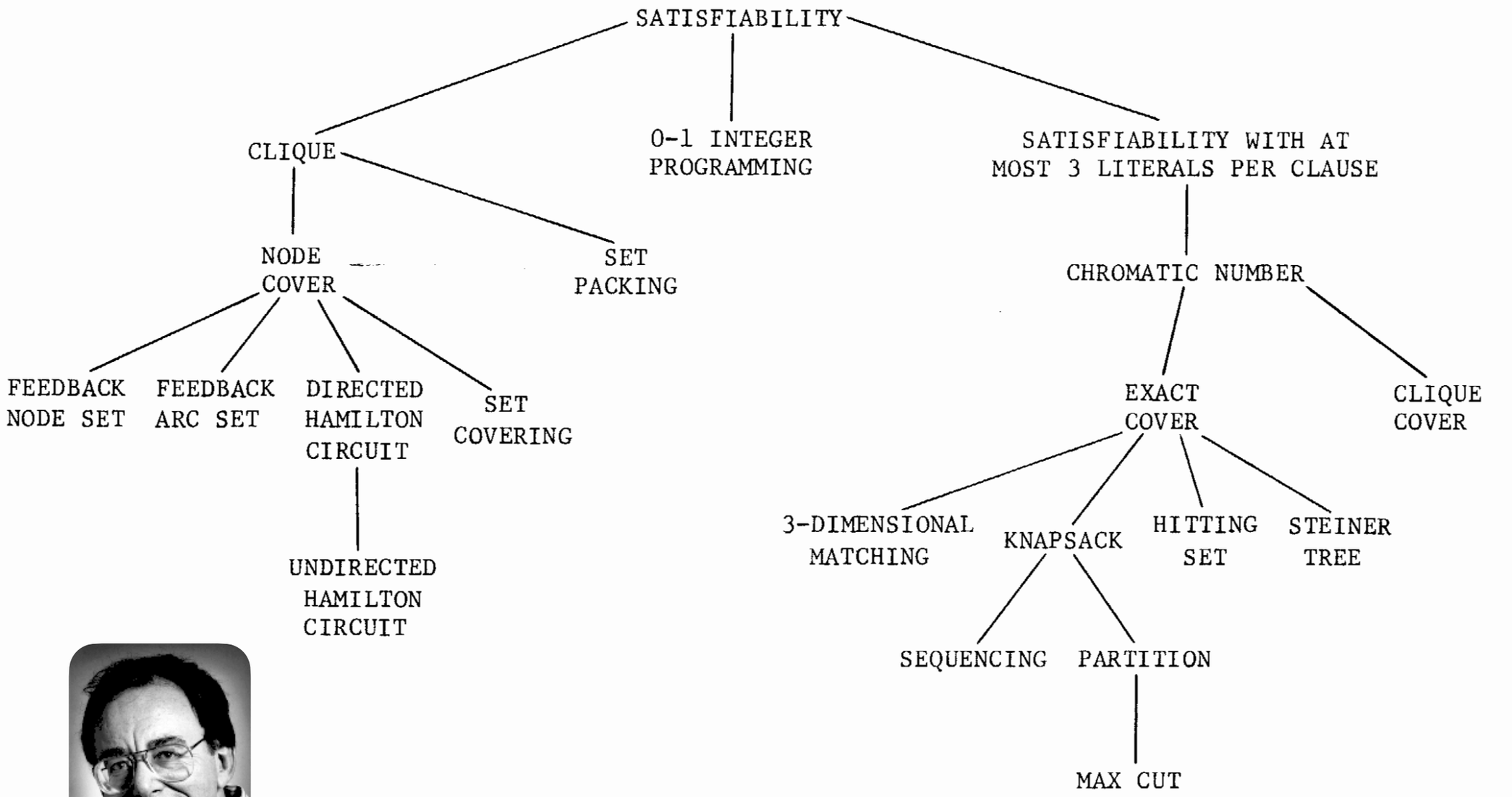


FIGURE 1 - Complete Problems

RICHARD M. KARP

Dick Karp (1972)
1985 Turing Award

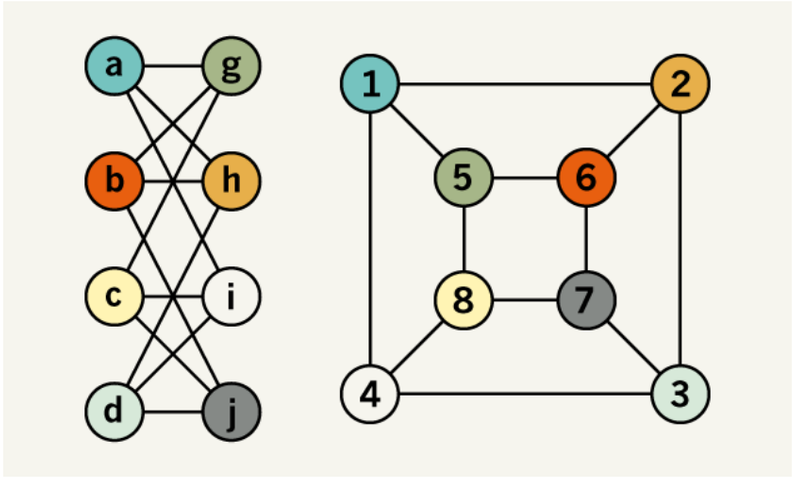
There have been thousands of NP-complete problems.

Problems Between P and NP-Complete

- There are problems likely to be "easier" than NP-complete problems, meanwhile no poly-time algorithms are found.

- Integer factoring: need to search for the answer, while the negative part of the decision problem is trivial (factorization always exists). Poly-time quantum factoring algorithm is found.

- Graph isomorphism:



Graph Isomorphism in Quasipolynomial Time

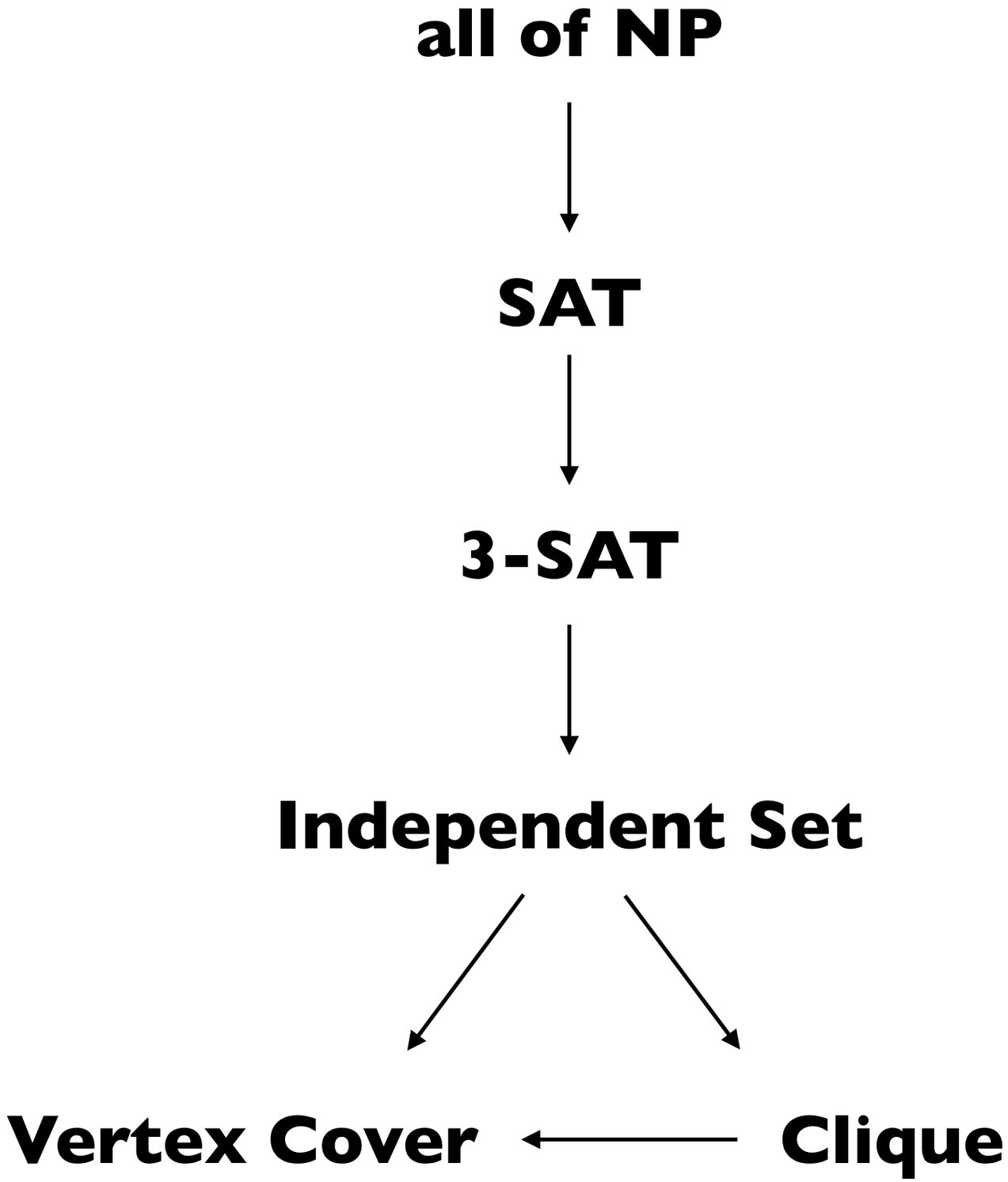
László Babai
University of Chicago

2nd preliminary version
January 19, 2016

Abstract

We show that the Graph Isomorphism (GI) problem and the related problems of String Isomorphism (under group action) (SI) and Coset Intersection (CI) can be solved in quasipolynomial ($\exp\left((\log n)^{O(1)}\right)$) time. The best previous bound for GI was $\exp(O(\sqrt{n \log n}))$, where n is the number of vertices (Luks, 1983); for the other two problems, the bound was similar, $\exp(\widetilde{O}(\sqrt{n}))$, where n is the size of the permutation domain (Babai, 1983).

Reductions for NP-Complete Problems



Satisfiability (SAT) Problem

Literal. A Boolean variable or its negation. x_i or $\overline{x_i}$

Clause. A disjunction of literals. $C_j = x_1 \vee \overline{x_2} \vee x_3$

Conjunctive normal form (CNF). A propositional formula Φ that is a conjunction of clauses. $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

SAT. Given a CNF formula Φ, does it have a satisfying truth assignment?

3-SAT. SAT where each clause contains exactly 3 literals (and each literal corresponds to a different variable).

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3 \right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3 \right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4 \right)$$

yes instance: x_1 = true, x_2 = true, x_3 = false, x_4 = false

3-SAT

3-Sat

Input: a 3-CNF formula

Output: whether the 3-CNF is satisfiable

- To satisfy a 3-CNF, we need to satisfy all clauses
- To satisfy a clause, we need to satisfy at least 1 literal
- Assignment $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$ satisfies $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

SAT to 3-SAT

Here's the trick for reducing SAT to 3SAT: given an instance I of SAT, use exactly the same instance for 3SAT, except that any clause with more than three literals, $(a_1 \vee a_2 \vee \cdots \vee a_k)$ (where the a_i's are literals and $k > 3$), is replaced by a set of clauses,

$$(a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ (\overline{y}_2 \vee a_4 \vee y_3)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k),$$

where the y_i's are new variables. Call the resulting 3SAT instance I'. The conversion from I to I' is clearly polynomial time.

Why does this reduction work? I' is equivalent to I in terms of satisfiability, because for any assignment to the a_i's,

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

SAT to 3-SAT

Here's the trick for reducing SAT to 3SAT: given an instance I of SAT, use exactly the same instance for 3SAT, except that any clause with more than three literals, $(a_1 \vee a_2 \vee \cdots \vee a_k)$ (where the a_i's are literals and $k > 3$), is replaced by a set of clauses,

$$(a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ (\overline{y}_2 \vee a_4 \vee y_3)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k),$$

where the y_i's are new variables. Call the resulting 3SAT instance I'. The conversion from I to I' is clearly polynomial time.

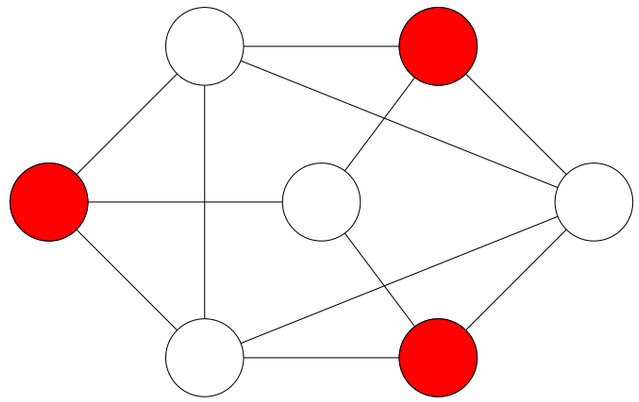
Why does this reduction work? I' is equivalent to I in terms of satisfiability, because for any assignment to the a_i's,

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Reduce a problem to its subset

Independent Set

Def. An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



Independent Set (Ind-Set) Problem

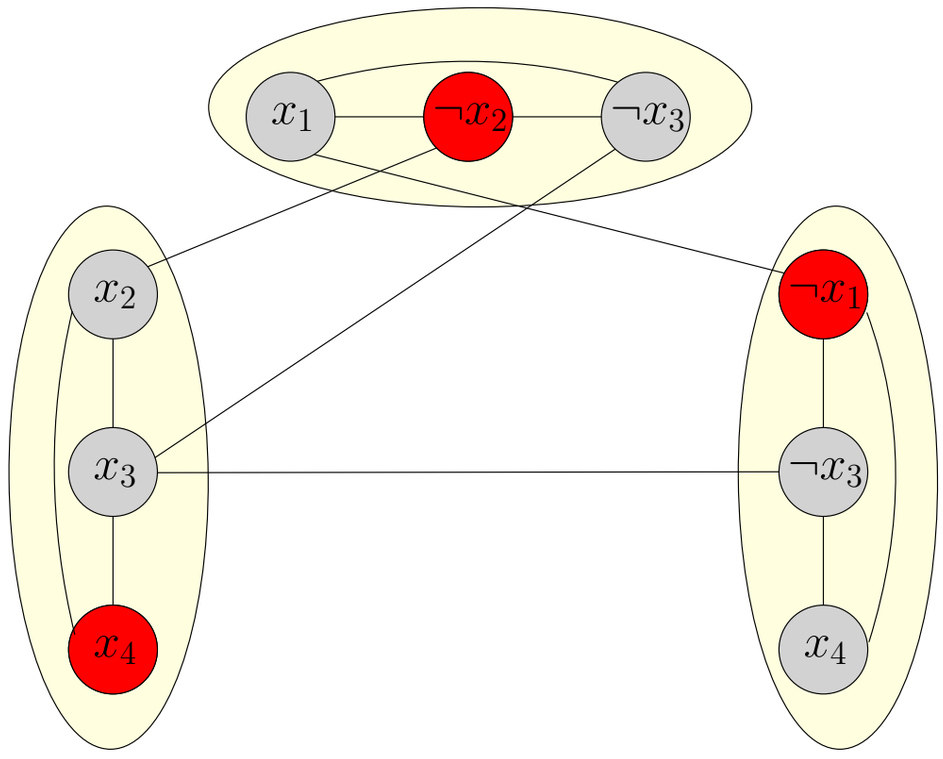
Input: $G = (V, E), k$

Output: whether there is an independent set of size k in G

3-SAT to Independent Set

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals
- Problem: whether there is an IS of size $k = \#$clauses



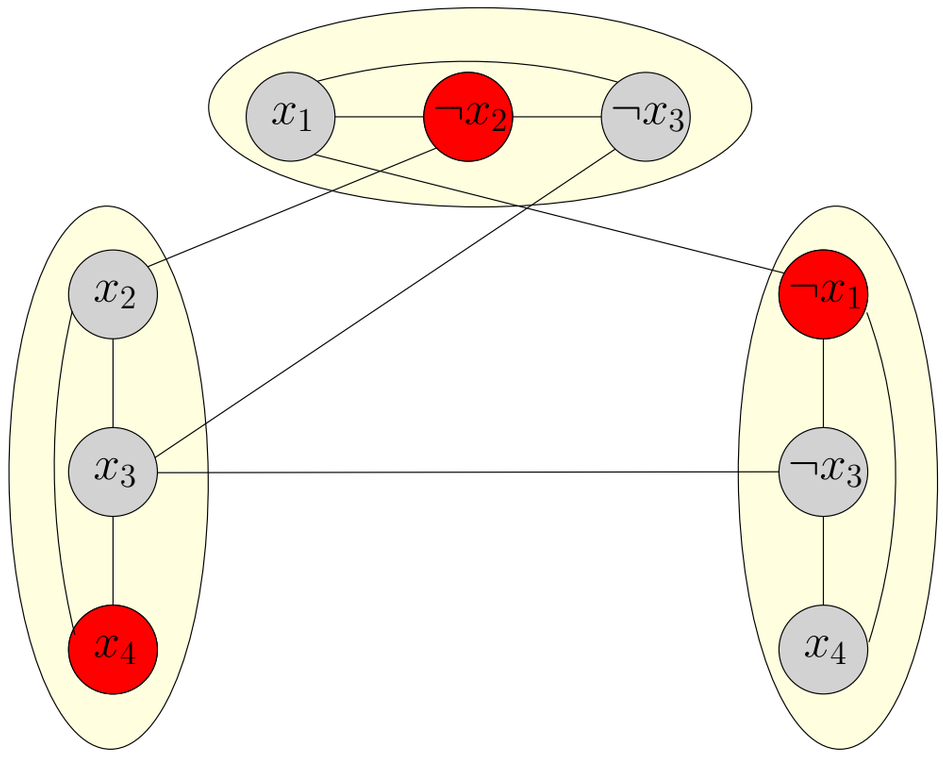
3-Sat instance is yes-instance $\Leftrightarrow$ Ind-Set instance is yes-instance:

- satisfying assignment $\Rightarrow$ independent set of size k
- independent set of size $k \Rightarrow$ satisfying assignment

3-SAT to Independent Set

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4)$

- A clause $\Rightarrow$ a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals
- Problem: whether there is an IS of size $k = \#$clauses



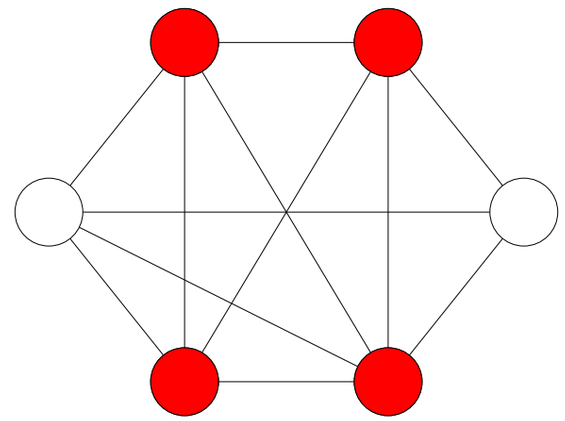
3-Sat instance is yes-instance $\Leftrightarrow$ Ind-Set instance is yes-instance:

- satisfying assignment $\Rightarrow$ independent set of size k
- independent set of size $k \Rightarrow$ satisfying assignment

Reduce between problems seem to differ much.

Clique

Def. A clique in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$



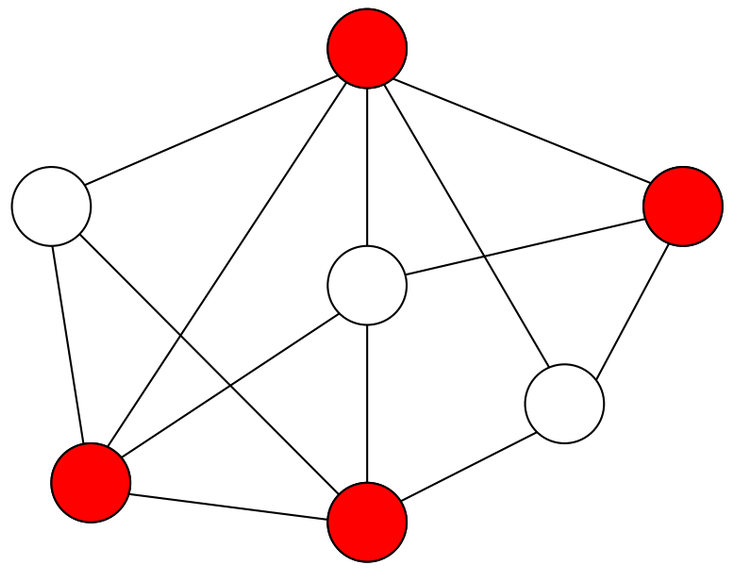
Clique Problem

Input: $G = (V, E)$ and integer $k > 0$,

Output: whether there exists a clique of size k in G

Vertex Cover

Def. Given a graph $G = (V, E)$, a vertex cover of G is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.



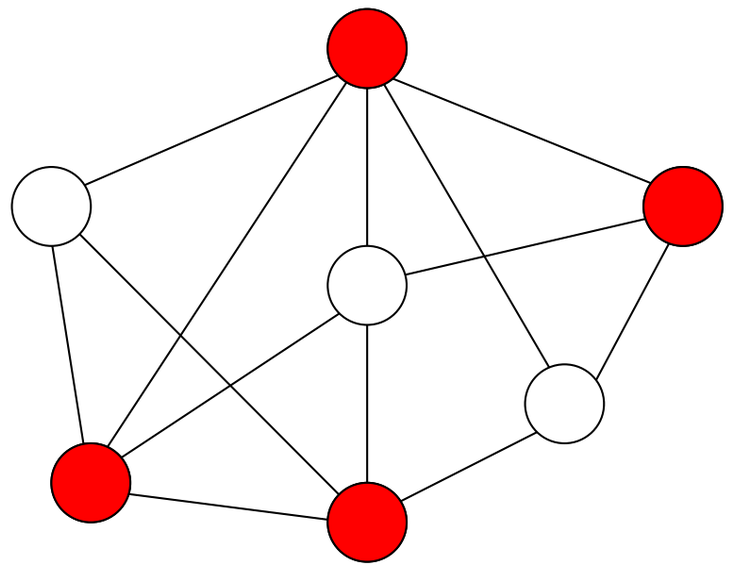
Vertex-Cover Problem

Input: $G = (V, E)$ and integer k

Output: whether there is a vertex cover of G of size at most k

Vertex Cover

Def. Given a graph $G = (V, E)$, a vertex cover of G is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.



Vertex-Cover Problem

Input: $G = (V, E)$ and integer k

Output: whether there is a vertex cover of G of size at most k

A special case of set cover
where each element is exactly in two sets.

Vertex Cover $=_P$ Independent Set $=_P$ Clique

Q: What is the relationship between Vertex-Cover and Ind-Set?

A: S is a vertex-cover of $G = (V, E)$ if and only if $V \setminus S$ is an independent set of G.

Def. Given a graph $G = (V, E)$, define $\overline{G} = (V, \overline{E})$ be the graph such that $(u, v) \in \overline{E}$ if and only if $(u, v) \notin E$.

Obs. S is an independent set in G if and only if S is a clique in $\overline{G}$.

〖**Example**〗 **Suppose that we already know that the clique problem is NP-complete. Prove that the vertex cover problem is NP-complete as well.**

❖ **Clique problem: Given an undirected graph G = (V, E) and an integer K, does G contain a complete subgraph (*clique*) of (at least) K vertices?**

CLIQUE = { <G, K> : G is a graph with a clique of size K }.

❖ **Vertex cover problem: Given an undirected graph G = (V, E) and an integer K, does G contain a subset V' ⊆ V such that |V'| is (at most) K and every edge in G has a vertex in V' (*vertex cover*)?**

VERTEX-COVER = { <G, K> : G has a vertex cover of size K }.

Proof: ① VERTEX-COVER ∈ NP

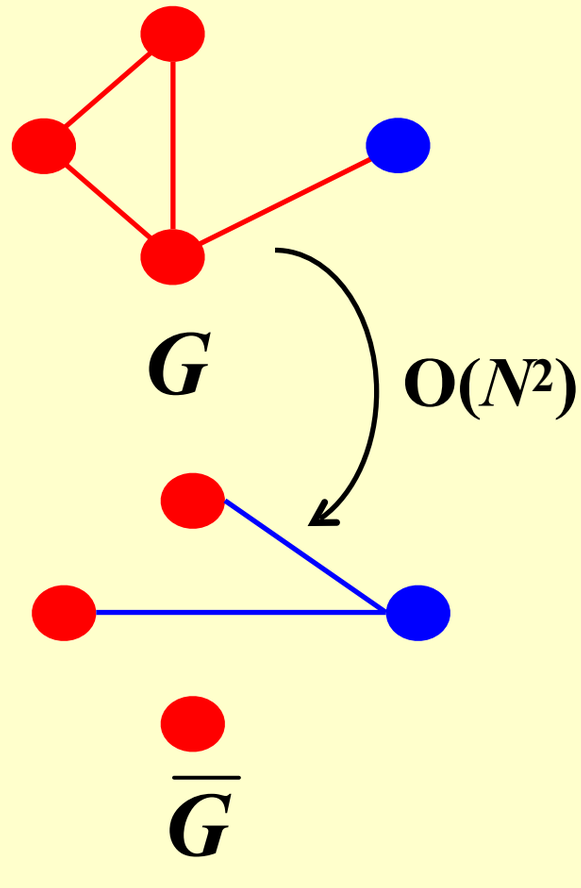
Given any *x* = <G, K>, take V' ⊆ V as the certificate *y*.

Verification algorithm: check if |V'| = K; check if for each edge (*u*, *v*) ∈ E, that *u* ∈ V' or *v* ∈ V'.

$O(N^3)$

Proof (con.): ② **CLIQUE $\leq_P$ VERTEX-COVER**

G has a clique of size K iff $\overline{G}$ has a vertex cover of size |V| – K.



⇒ G has a clique V' ⊆ V of size K

Let (u, v) be any edge in $\overline{E}$ ➡

At least one of u or v does not belong to V'

At least one of u or v does belong to V-V'

Every edge of $\overline{G}$ is covered by a vertex in V-V'

Hence, the set V-V', which has size |V| – K forms a vertex cover for $\overline{G}$

⇐ $\overline{G}$ has a vertex cover V' ⊆ V of size |V| – K

For all $u, v \in$ V, if $(u, v) \notin$E , then $u \in$V' or $v \in$ V' or both.

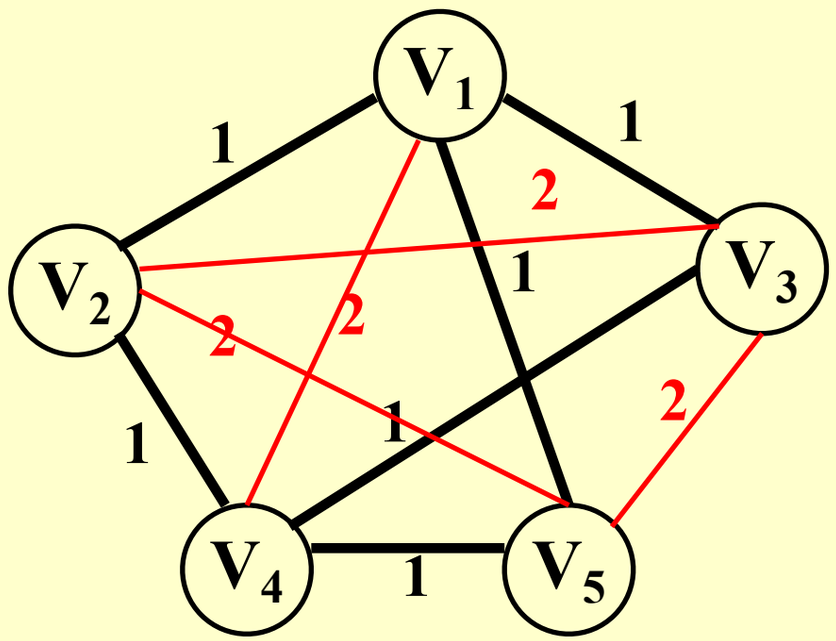
For all $u, v \in$ V, if $u \notin$V' AND $v \notin$ V', then $(u, v) \in$ E.

V-V' is a clique and it has size |V|–|V'| = K. ∎

〖Example〗 Suppose that we already know that the Hamiltonian cycle problem is NP-complete. Prove that the traveling salesman problem is NP-complete as well.

- ❖ Hamiltonian cycle problem: Given a graph G=(V, E), is there a simple cycle that visits all vertices?
- ❖ Traveling salesman problem: Given a complete graph G=(V, E), with edge costs, and an integer K, is there a simple cycle that visits all vertices and has total cost ≤ K?

Proof: TSP is obviously in NP, as its answer can be verified polynomially.



K = |V|

G has a Hamilton cycle iff G' has a traveling salesman tour of total weight |V|.

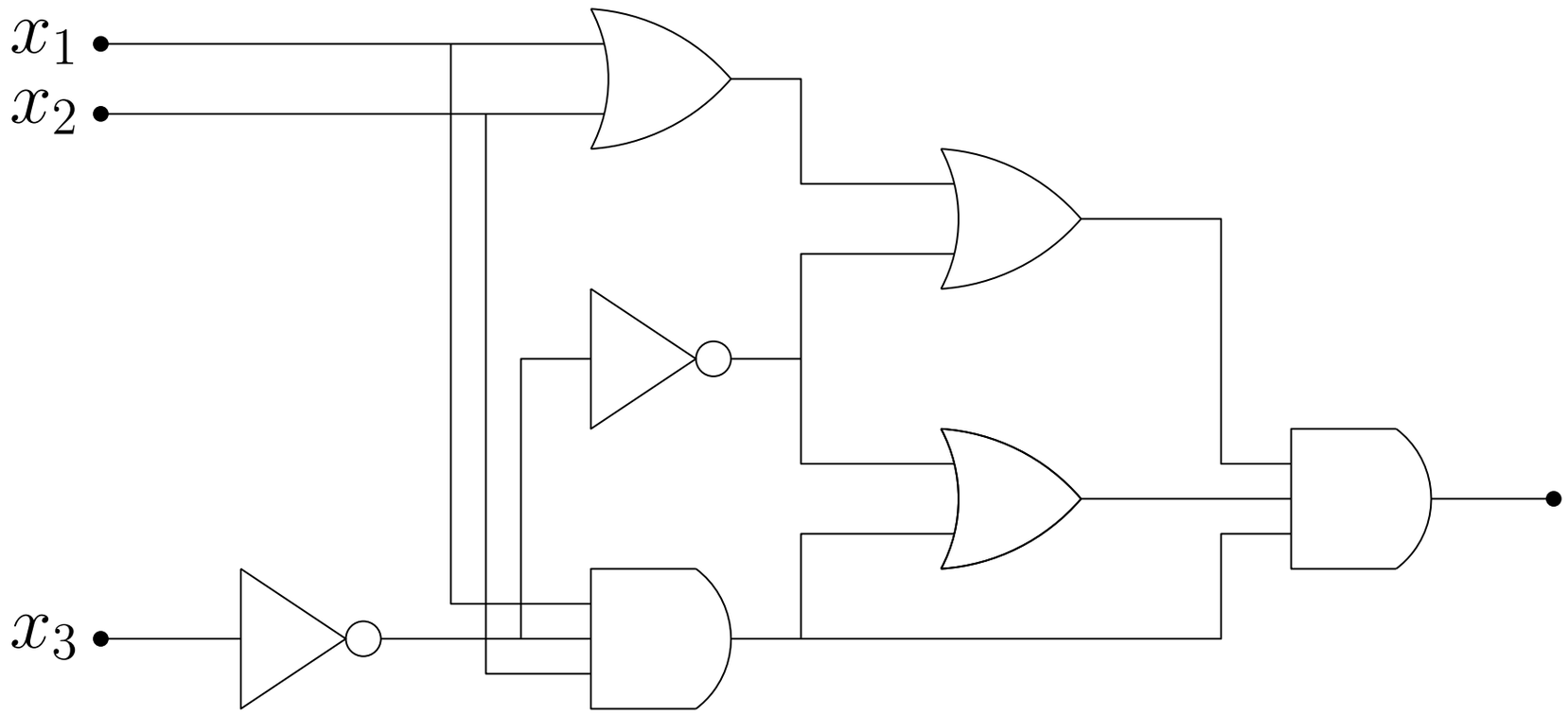
✓

All NP Problems to SAT

Circuit Satisfiability (Circuit-Sat)

Input: a circuit

Output: whether the circuit is satisfiable

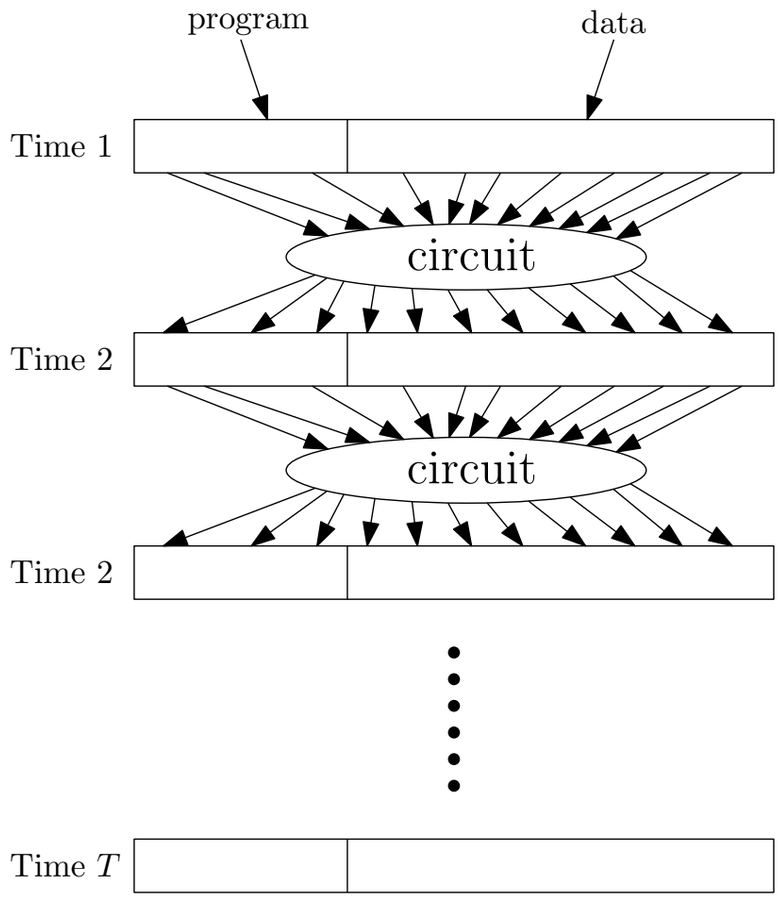


Theorem Circuit-Sat is NP-complete.

All NP Problems to SAT

- key fact: algorithms can be converted to circuits

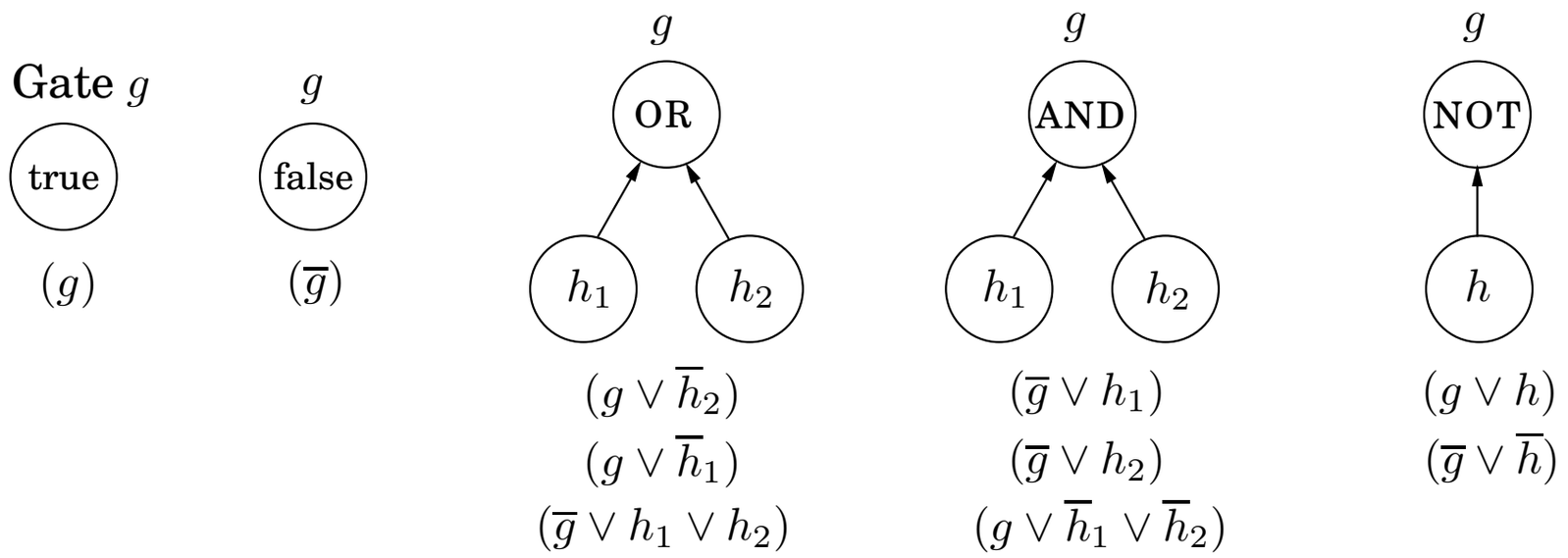
Fact Any algorithm that takes n bits as input and outputs $0/1$ with running time $T(n)$ can be converted into a circuit of size $p(T(n))$ for some polynomial function $p(\cdot)$.



- Then, we can show that any problem $Y \in$ NP can be reduced to Circuit-Sat.

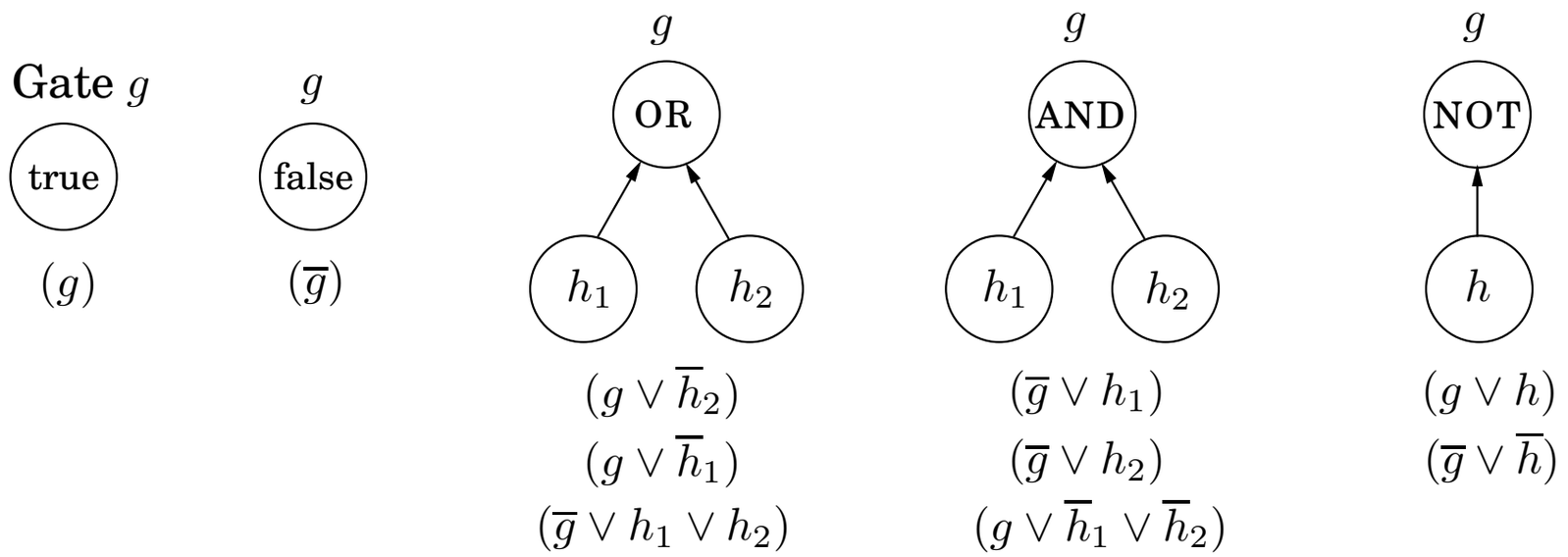
All NP Problems to SAT

Going in the other direction, CIRCUIT SAT can also be reduced to SAT. Here is how we can rewrite any circuit in conjunctive normal form (the AND of clauses): for each gate g in the circuit we create a variable g, and we model the effect of the gate using a few clauses:



All NP Problems to SAT

Going in the other direction, CIRCUIT SAT can also be reduced to SAT. Here is how we can rewrite any circuit in conjunctive normal form (the AND of clauses): for each gate g in the circuit we create a variable g, and we model the effect of the gate using a few clauses:



SAT is a special case of circuit-SAT with simpler circuit structure.
While actually circuit-SAT can be reduced to SAT with
the similar technique from SAT to 3-SAT.

Outline: Intractability

- Computability and computation complexity
- P, NP, and Co-NP
- NP-completeness
- Take-home messages

Take-Home Messages

- Computability: Everything computable can be computed by Turing machines, while there are uncomputable functions for TMs.

> One famous problem of this sort is an arithmetical version of SAT. Given a polynomial equation in many variables, perhaps
>
> $$x^3yz + 2y^4z^2 - 7xy^5z = 6,$$
>
> are there integer values of x, y, z that satisfy it? There is no algorithm that solves this problem. No algorithm at all, polynomial, exponential, doubly exponential, or worse! Such problems are called *unsolvable*.

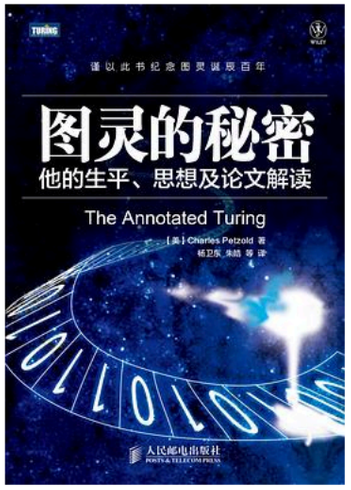
Hilbert's tenth problem

- P, NP, Co-NP: Complexity classes of decision problems.
  - Class P: poly-time computable
  - Class NP and Co-NP: poly-time (positively or negatively) verifiable
- NP-complete problems: Hardest problems in NP, all NP problems reduce to them. Don't have poly-time algorithm unless P=NP.

Thanks for your attention!
Discussions?

Further Readings

图灵的秘密



更新图书信息或封面

作者: Charles Petzold
出版社: 人民邮电出版社
出品方: 图灵新知
副标题: 他的生平、思想及论文解读
原作名: The Annotated Turing: A Guided Tour Through Alan Turing's Historic Paper on Computability and the Turing Machine
译者: 杨卫东
出版年: 2012-11
页数: 344
定价: 69.00元
ISBN: 9787115282149

豆瓣评分

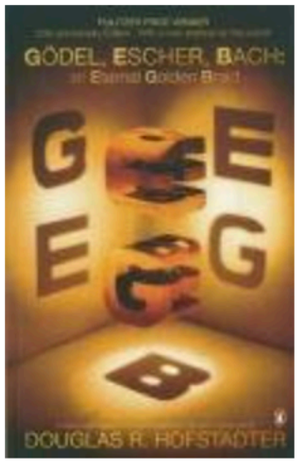
8.7 477人评价

5星 55.1%
4星 32.9%
3星 8.8%
2星 2.1%
1星 1.0%

好友评分

6.0 1人评价

Godel, Escher, Bach



更新图书信息或封面

作者: Douglas R Hofstadter
出版社: Penguin
副标题: An Eternal Golden Braid
出版年: 2000-03-30
页数: 824
定价: GBP 18.99
装帧: Paperback
丛书: Basic Books
ISBN: 9780140289206

豆瓣评分

9.4 269人评价

5星 79.2%
4星 14.1%
3星 6.3%
2星 0.0%
1星 0.4%

好友评分

10.0 1人评价

Reference

Algorithms (DPV): Chap. 8.

Algorithm design: Chap. 8

Introduction to Algorithms (4th Edition): Chap. 34.

Slides for intractability from Kevin Wayne: https://www.cs.princeton.edu/~wayne/kleinberg-tardos/

Shi Li's lecture: https://tcs.nju.edu.cn/shili/courses/2024spring-algo/slides/NPC.pdf

Turing machine descriptions from computational complexity book by Arora and Barak (Chap. 1): https://theory.cs.princeton.edu/complexity/book.pdf